DESIGN AND SYNTHESIS
OF A REAL-TIME CONTROLLER
FOR AN UNMANNED AIR VEHICLE

Peter M. Hoffman
Lieutenant, United States Coast Guard
B.S., United States Coast Guard Academy, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
and
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL

September 1994

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
Design and Synthesis of a Real-Time Controller for an Unmanned Air Vehicle (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Hoffman, Peter M.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Naval Postgraduate School is developing an vertical take-off and landing (VTOL) unmanned air vehicle (UAV) that can transition to horizontal flight, once airborne, in order to take advantage of the improvements in speed, range, and loiter time that horizontal, fixed-wing flight provides. This research investigates the design requirements of the central controlling device for that UAV, including the specific problems of defining the necessary hardware components and developing software for executive control. First, hardware requirements needed to be determined. By exploring the general operational requirements of the UAV and taking into account space and weight limitations, a hardware suite was selected which could provide adequate functionality to replace the human traits of a pilot. In order to provide "awareness" of the operational environment, motion sensors, navigation equipment, and communication equipment was required. Controllable servo motors were necessary to move control surfaces appropriately. Computer hardware, necessary to provide system intelligence, was selected in order to interoperate with the other hardware. Next, a Real-Time Executive (RTE) software program was designed to provide the functionality and coordination of all hardware components. Device drivers for each component were developed, and overall coordination was planned using a Yourdon style essential model. Periodic interrupts were used to control execution time. Last, the specifications and configuration of all hardware components were completely documented, and the operation of the RTE program is fully explained. From this understanding of the overall control system, future development can continue, resulting in a more effective and efficient UAV design.

14. SUBJECT TERMS
Unmanned Air Vehicle, UAV, Remote Control, Controller, Microprocessor, Real-Time Executive, DOS Interrupts, GPS, IMU, Datalink

15. NUMBER OF PAGES
122

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

ABSTRACT

The Naval Postgraduate School is developing an vertical take-off and landing (VTOL) unmanned air vehicle (UAV) that can transition to horizontal flight, once airborne, in order to take advantage of the improvements in speed, range, and loiter time that horizontal, fixed-wing flight provides. This research investigates the design requirements of the central controlling device for that UAV, including the specific problems of defining the necessary hardware components and developing software for executive control.

First, hardware requirements needed to be determined. By exploring the general operational requirements of the UAV and taking into account space and weight limitations, a hardware suite was selected which could provide adequate functionality to replace the human traits of a pilot. In order to provide "awareness" of the operational environment, motion sensors, navigation equipment, and communication equipment was required. Controllable servo motors were necessary to move control surfaces appropriately. Computer hardware, necessary to provide system intelligence, was selected in order to interoperate with the other hardware. Next, a Real-Time Executive (RTE) software program was designed to provide the functionality and coordination of all hardware components. Device drivers for each component were developed, and overall coordination was planned using a Yourdon style essential model. Periodic interrupts were used to control execution time. Last, the specifications and configuration of all hardware components were completely documented, and the operation of the RTE program is fully explained. From this understanding of the overall control system, future development can continue, resulting in a more effective and efficient UAV design.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY OF TERMS

A/D   Analog to Digital

ASCII   American Standard Code for Information Interchange

BIOS   Basic Input Output System

CMOS   Complementary Metal Oxide Semiconductor

COTS   Commercial Off-The-Shelf

CPU   Central Processing Unit

CTS   Clear to Send Signal

D/A   Digital to Analog

DCD   Data Carrier Detect Signal

DR   Dead Reckoning (Position)

DRAM   D-Flip Flop Random Access Memory

DSR   Data Set Ready Signal

DTE   Data Terminal Equipment

DTR   Data Terminal Ready Signal

EOI   End of Interrupt Signal

FMU   Flight Management Unit

GPS   Global Positioning System

IDE   Integrated Development Environment (used in Borland compiler)

IMU   Inertial Measurement Unit

INS   Inertial Navigation System

I/O   Input/Output

IRQ   Interrupt Request Signal

ISR   Interrupt Service Routine

LKP   Last Known Position

Modem   Modulator-Demodulator

MS-DOS   Microsoft Disk Operating System

PANDL   Programming structure containing Pointer to a buffer AND the buffer Length

PF   Position Fix

PIC   Peripheral Interrupt Controller

PIT   Programmable Internal Timer

RPV   Remotely Piloted Vehicle

RTC   Real Time Clock

RTE	Real-Time Executive
RTS	Request to Send Signal
STC	System Timing Controller
TNC	Terminal Node Controller
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver and Transmitter
UAV	Unmanned Air Vehicle or Unmanned Aerial Vehicle
UHF	Ultra-High Frequency
VTOL	Vertical Take-Off and Landing

I. INTRODUCTION

Control is defined as "the right or prerogative of determining or governing" [Ste73]. The control of any moving entity requires a correct determination of the current state of that entity followed by the careful governing of actions taken to correct any deviation from a given desired state. In an aircraft, this function is usually done by a human being. When removing the pilot from the aircraft, this control falls to a coordinated system of hardware and software called a *controller*. This controller must be built around some source of intelligence, whether it is remotely linked to a human, or intelligent in its own right. In addition, the controller must have the capability to create and govern the necessary changes in the aircraft's state. Since the controller's roles are so varied and yet interrelated, there are many design options to consider -- the permutations of which comprise many possible, workable solutions.

A. RESEARCH OBJECTIVE

This research examines the design and synthesis of a controller for an unmanned air vehicle (UAV). The primary research question is "What is required to build a central controller for an UAV?" A central controller is differentiated from any other auxiliary controllers that may be on board in that the central controller governs the actual flight control of the aircraft. This primary research question encompasses both hardware and software requirements, and leads to additional questions:

- What sensors will be used to provide information about the state of the aircraft, in the absence of human senses?
- What is the form and limitations of the data provided by these sensors?
- What aerodynamic surfaces are available to control the aircraft?
- What devices are available to move the control surfaces, and what signals are required to cause these devices to move those surfaces?
- How must the movement of these control surfaces be coordinated to control the aircraft in its six degrees of freedom?
- Does the UAV need to communicate with any external facilities?
- If the UAV does need to communicate with external facilities, how should this be accomplished?
- What are the format and buffering requirements for this communication link?
- What other input and output (I/O) of data is required?
- What are the timing constraints for all command, control, and communication operations?
- What hardware is necessary to achieve this functionality?
- What are the software requirements to interact with the chosen hardware?

With these questions in mind, the parameters of operation and the system requirements for a UAV controller are investigated. The specifications and configuration of the hardware and software used are fully

documented. The goal of this research project is a functional UAV controller, including a comprehensive Real-Time Executive software program fully integrated with the necessary hardware.

B.   PREVIOUS AND CONCURRENT RESEARCH

Some of these questions have already been answered. This multi-faceted project is being developed in several previous and concurrent research programs, bringing together disciplines from Aeronautical Engineering; Mechanical Engineering; Electrical Engineering; Command, Control and Communications; and Computer Science. The airframe was aerodynamically analyzed and modified by Stoney [Sto93]. Significant to this research was the addition of a canard on the front of the aircraft, which included two additional control surfaces. The servo motors used to move these and other control surfaces were examined by Merz [Mer92] and Moran [Mor93]. They developed a core control system for the aerodynamic control vanes. Two datalinks have been developed to facilitate the transfer of data to and from a ground station. Reichert [Rei93] designed a wide-band UHF system and Bess [Bes94] tested a spread spectrum datalink. For navigation, Twite [Twi94] developed a differential GPS navigation system, and Hallberg [Hal94] investigated the inertial measurement unit (IMU) which was used. Marquis [Mar93] designed a complementary Kalman filter to blend the outputs of these two sensors, and the control algorithm has been worked on by Davis [Dav92], Kuchenmeister [Kue93], Hallberg [Hal94], and Moats [Moa94].

C.   EXECUTIVE SUMMARY

This document consists of five chapters, including this Introduction. Chapter II provides a generic overview of the controller, discusses required functionality, and develops an essential model of this functionality in order to better understand the constraints and criteria under which it must operate. Chapter III fully documents the specifications and configuration of all hardware used for the controller. It is intended to provide information in sufficient detail such that a new user would understand how to duplicate the existing hardware system upon receipt using similar equipment. Chapter IV fully documents the RTE software written for this project. This includes compiler information to enable a new user to recreate the software development environment under which the software was created. Chapter V delineates the conclusions drawn from this research and provides recommendations for future improvements to the system. Appendix A contains a listing of the source code for the RTE. Appendix B contains a glossary listing of all variables used in the RTE program. Finally, Appendix C contains the manufacturer's technical data sheets for the hardware used for the controller.

This research project will tie together the many individual sub-systems that have been designed for the UAV and provide the means for their operational use and coordination. Additionally, this research stands as proof-of-concept for UAV technology, especially for vertical take-off and landing (VTOL) aircraft that can transition to horizontal flight. This characteristic makes it especially useful for shipboard deployment, which will benefit not only the Department of Defense, but also the U.S. Coast Guard by providing a cost-effective and fatigue-resistant solution to many airborne missions, such as Search and Rescue and Law Enforcement.

II. BACKGROUND

This chapter provides background information detailing the objectives introduced in the last chapter. In this chapter, a generic overview of a controller is provided, design considerations are discussed, and a Yourdon style essential model [You89] is developed. Within this model, parameters and priorities unique to this system are investigated. Since it is a real-time system, timing and intra-process communications are determined. This analysis results in a more effective and efficient controller design.

A.   SYSTEM OVERVIEW

In order to control the aircraft, a controller must have access to all information pertaining to the operation of the airframe, including:

- Present and desired geographic position
- Present and desired altitude
- Present and desired airframe attitude (in 3 dimensions)
- Present airframe acceleration (in 3 dimensions)
- Present state of power plant, including throttle and fuel state
- Present state of payload equipment and desired operation of such equipment

Then, a controller must be able to properly manipulate this information, develop signals to modify the physical configuration of the aircraft's control surfaces, and effect necessary data communications. To accomplish these results, a controller must maintain control over the following:

- Signals that control each of the aircraft's control surfaces
- Signals that control the power plant (throttle)
- Signals that control the payload equipment
- Communication channels with ground control and/or monitoring stations

In a digital controller, these signals are processed by a set of software algorithms, interacting with specialized hardware. This set of algorithms comprises a Real-Time Executive program; the hardware includes sensors, servos, and communication equipment. As detailed by Moran [Mor93] and in Chapter III, the Archytas airframe uses the following components to meet these requirements:

1.   Aircraft Sensors

The UAV needs specially designed sensors which can generate signals in response to position, posture and acceleration of the aircraft. Among the sensors with which the processor must interact are:

- Global Position System (GPS) Satellite Receiver
- Inertial Navigation System (INS) Instruments
- Other (Non-INS) Flight Instruments
- Fuel Sensor

The GPS receiver yields a time stamp and a 3-dimensional position, including latitude, longitude, and altitude. The INS instruments include accelerometers that measure linear acceleration in each of the three dimensions, roll-rate sensors that measure angular velocity about each of the three coordinate axes, gyros that indicate aircraft heading, and vertical inclinometers that measure the angle of tilt from vertical. Non-INS instruments include a pitot tube airspeed indicator and a barometric altimeter. These sensors are "strapped down" which means they are fixed in alignment with the *body coordinate system*, the 3-dimensional coordinate system that uses the body of the aircraft as a reference. They detect acceleration, velocity or displacement in a certain direction, in reference to the aircraft itself. To be useful, these signals must be converted to the coordinate system of the environment surrounding the aircraft, called the *inertial coordinate system*. This conversion involves only a basic matrix rotation, but can demand significant processing time.

2.   Aircraft Components

In addition to the sensors, the controller has several extrinsic components with which the processor must interact. These include:

- Pulse-Width Modulated Servos
- Communications Link
- Payload Equipment

The servos are electric motors that position aircraft controls as determined by the length of a received square pulse. The communication link is usually a radio datalink transmitting digital data. It provides two-way communication with the UAV, uplinking control information (commands) from the ground control station, and downlinking flight instrument data to a ground monitoring station. Payload equipment includes cameras, radar, and other sensors not necessary for flight control, but used to complete the assigned mission.

3.   System Block Diagram

The components and sensors are merged together to form an integrated flight control system, as shown in Figure II-1. The most basic UAV controller must be able to perform the following functionality:

- Receive digital sensor inputs
- Perform analog to digital conversion of all analog sensor inputs
- Provide digital filtering for sensor data
- Accept and process joystick and/or waypoint pilot commands from the uplink
- Using input above, calculate control functions and determine control surface positions.
- Generate appropriate command pulses and transmit them to servos.
- Relay necessary posture and navigation information to ground through downlink

The software components of this system are the Kalman filter and the control algorithms. The Kalman filter is an algorithm that smooths the sensor data to provide the most reliable source of data input, based on the varying accuracy of given sensors over different periods of time. The control algorithms are based on aerodynamic properties specific to the UAV. These algorithms compare the posture and position of the aircraft to the desired posture and position, and determine what corrective actions must be taken. These software functions, in addition to the transfer of all data and signals coming from or going to the central processor (shown as arrows to and from the shaded area in Figure II-1), comprise the Real-Time Executive - - the software component of the controller.
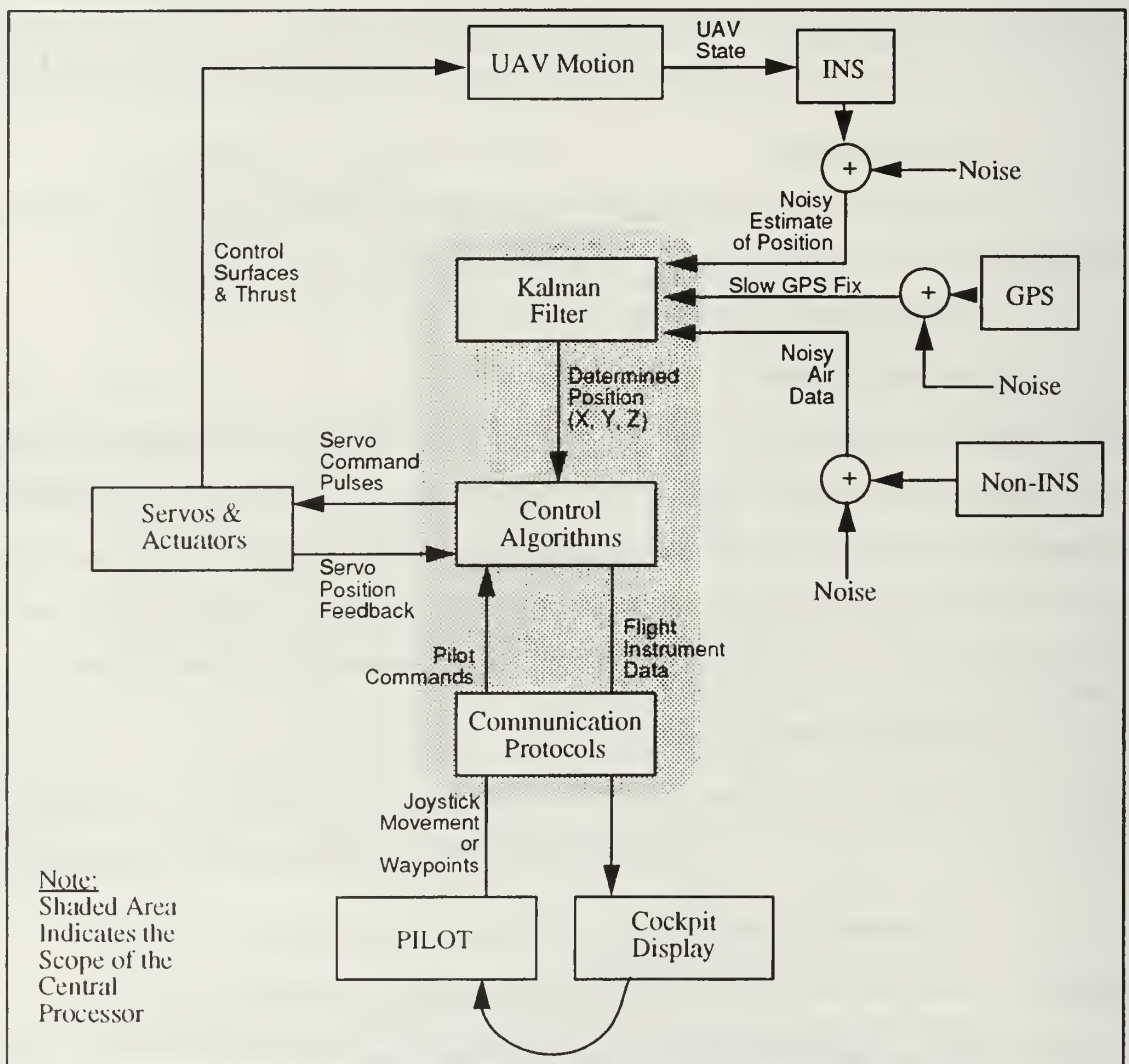
Figure II-1: System Block Diagram

4. Assumptions

Since this project is based on the work of several students working concurrently, the scope of the controller's functionality for this research is limited by several assumptions. All of these assumptions can be substantiated once the controller is fully completed and tested on the aircraft in flight.

a. *Filtering and control algorithms exist as system callable subroutines.*

The filtering and control algorithms are specialized aeronautical engineering routines which involve additional research, such as linear quadratic filter design and wind tunnel testing, which is being accomplished by other students. For this project, it is assumed that the resulting routines will exist and can execute within the allotted time constraints.

b. *Input and output (I/O) will not require central processing resources.*

Digital I/O will take place through eight 25-pin RS-232 serial ports. These ports reside on a separate circuit board that has its own small processor. Although I/O must be initiated by a subroutine call from the main controller program, the processing required to execute and control the data flow is handled by this sub-processor and so will not require central processing resources. This form of parallel processing greatly reduces the amount of processing time required for this function.

c. *Air to ground communications will not require central processing resources.*

The radio data link hardware on hand conducts its own handshaking, parity checking, and other data communication functions. Since it will connect to the I/O ports, this function is actually twice removed from the controller itself. Even with an assumed 15% protocol overhead, the datalink is expected to have sufficient throughput to avoid becoming a bottleneck in the communication process, without requiring central processing resources.

d. *The sensors' data stream will be fast enough to provide fresh data for every cycle.*

It is imperative that the digital data from the Inertial Measurement Unit (IMU) is current and complete every control cycle. Additionally, sensor data which is not digital must first be converted from analog to digital (A/D) through a sampling A/D circuit board. This A/D process is done in hardware and will not affect the performance of the controller. However, if this A/D process takes too long, the most recent data from the A/D card may not yet be available when read by the controller. It is assumed that current and complete sensor data will be available when needed for each control cycle.

e. The controller's processing speed will be sufficiently fast to perform all functions.

No matter how fast a processor is, it has a limited throughput. It is assumed that the CPU can perform all required functions in the allotted real-time interval. If this assumption proves false, the present processing speed of 33 MHz will have to be upgraded.

B. DESIGN CONSIDERATIONS

In the design of a real-time system, it is important to understand the constraints, structures, and parameters of the system, as well as other design choices that are available to the designer. It is the combination of these design choices that determines the successfulness of the resulting system. This section details the constraints imposed by a real-time system, lists options for programming structure, and discusses considerations for the selection of a programming language and fault tolerance measures.

1. Real-Time System Constraints

The term *real-time* covers a wide range of systems; however, all systems share a common feature where results of some kind are demanded by timing deadlines imposed by the environment outside the system [Sav85]. As time marches relentlessly onward, all system and subsystem responses must fit within their allotted time frames. A real-time system can also be described as reactive or embedded. *Reactive systems* are those which have some ongoing interaction with their environment. *Embedded systems* are those used to control specialized hardware in which the computer system is installed. Since the UAV controller will continuously monitor and interact with the position and posture of the aircraft within its environment and will also simultaneously control specialized hardware, it is both a reactive and an embedded system.

It can be argued that all practical systems are real-time systems. Even a word-processing system must respond to user commands within a reasonable amount of time (e.g. 1 second), or it will become torture to use. Most literature refers to such systems as *soft* real-time systems -- systems where performance is degraded but not destroyed by failure to meet response time constraints. Systems in which failure to meet response time constraints leads to potential complete system failure are called *hard* real-time systems. Under these definitions, the UAV controller is a hard real-time system under which missing a deadline could lead to complete loss of control. Therefore deadlines, once established, become constraints inside which the system must operate completely.

To meet these constraints, three measures of time, when applied to real-time systems, must be carefully managed: response time, survival time, and throughput [Sav85]. Each is defined below.

a. Response Time

Response time is the time the computer takes to recognize and respond to an external event. This is the most important time measurement in control applications. If events are not handled in a timely fashion, the system may literally go out of control. The UAV must not only respond to pilot commands, but must continually monitor feedback signals from the servos and inertial navigation equipment to determine if the response resulted in the correct, desired effect. Experimental evidence suggests that a total response time from pilot input to final movement of control surfaces cannot exceed 100 msec without loss of positive flight control by the pilot [Kan93].

b. Survival Time

Survival time is the time span during which data is available to be read. Since flight data is stored in a buffer, the data may be read at any time that the buffer is not being written to. Read/write cycles, therefore, must be sufficiently offset such that the reading and writing of the same data will not happen simultaneously. The next consideration, then, is the validity of the data. Since the aircraft is anticipated to move at speeds of up to 150 kts, it is important to have the latest sensor data available for every control cycle.

c. Throughput

Throughput is the total number of events which the system can handle in a given time period. For example, a communication controller may have a throughput expressed in characters per second. Since a large amount of data must be relayed to the ground, the radio datalink must not be allowed to become a bottleneck which could slow the central processor. The data stream must also be managed to flow evenly, so that there are not bursts of data in excess of the channel capacity interspersed among long periods of under-utilized capacity. In analog to digital (A/D) conversions, the bandwidth of the digital signal (equal to the product of the sampling rate and the sampling width in bits) is usually much greater than the bandwidth of its analog counterpart. Accordingly, the UAV A/D processes must be fast enough to provide fresh, accurate data at the frequency needed by the controller.

2. Structure Taxonomy

The simplest kind of software structure for a real-time system is a *polling loop*. The program examines (polls) each of its inputs in turn to determine whether an event has occurred which requires a response. This structure would be sufficient for the UAV if all polling was done at the same frequency. Since

this is not the case, a more complex, event-driven structure is needed. Event driven systems have three main types: interrupt driven, multi-tasking, or multi-processing (multiple processors) [Sav85].

a. Interrupt Driven

In an interrupt driven system, counters are used to keep track of inter-process timing, generating an interrupt when it is time to begin a new cycle. At the occurrence of an interrupt, the controller saves its current state on the stack and jumps to the appropriate interrupt service routine (ISR). If the timer generates an interrupt at regular intervals, and if the ISR is replaced by the control loop routine, it can be assured that the control loop will be executed regularly and consistently. However, for this to work, the execution time of the control loop must not exceed the interrupt time interval, or the subsequent ISR execution will interrupt the current ISR execution, resulting in a backlog of pending processes on the stack and, eventually, a system crash. To get the most from each control cycle, the task load resulting from each cycle should be kept as level as possible.

b. Multi-tasking

Task management could be delegated to the operating system if a multi-tasking environment was adopted. Multi-tasking, however, requires a large amount of processor overhead and brings with it its own unique problems when multiple tasks are forced to work with the same data. Since multi-tasking is not available on the present operating system, this option was not considered.

c. Multi-processing

To a certain degree, the UAV will have multiple processors, as mentioned in the assumptions. The benefit of multi-processing is in the ability to take the processing load for these functions off the central controller and to have them performed by a processor that is specially designed for that task. It is important to ensure that these auxiliary processors can access the same data buffers and to configure memory access such that no two can read/write the same data simultaneously.

3. Programming Language

"C" was chosen as the programming language for this project since it is almost unique among the high-level compiler languages in that it does not (usually) come in between the programmer and the machine. If something can be done in assembly language, there is usually a way to express it in C [Sav85]. For example, in C one can directly manipulate machine registers. I/O addresses can be written to directly. Interrupt handling is also possible. Interrupt vectors can be inspected and modified, and BIOS interrupts can

be executed by a system call. Memory allocation can be directly controlled, and bit-level programming is possible. For any applications that cannot be completed in C, assembly language could be used.

4. Fault Tolerance

Fault tolerance, or system robustness, is the ability to recover from errors or system failures. With an interrupt-driven system, provisions must be made for detecting and covering for a missed deadline. This can be done by spatial or temporal methods [Lap92]. *Spatial* fault tolerance includes redundant hardware and software systems. *Temporal* fault tolerance involves careful design of algorithms to compensate for missed deadlines. Since hardware on the UAV must be kept to a minimum, most of this redundancy must occur in the software. Although this will add to the overall complexity and detract from the overall efficiency of the system, the nature of the mission requires this redundancy. In addition, execution time for the fault tolerance overhead must not cause timing constraint violations in an otherwise correct system [Nel92]. Timing, execution and resource constraints dictate the following provisions for any module, regardless of the language that is used [Sta88]:

- Modules should have predetermined and bounded execution times.
      (recursion and loops must be used carefully)
- The use of dynamic structures should be controlled.
- For predictable system behavior, provision should be made for all known types of exceptions.

In addition to the normal programming exceptions, the RTE for the UAV also has to deal with external malfunctions, such as equipment failure, manual system reset, or loss of communication with the ground station.

C. AN ESSENTIAL MODEL

From the foregoing descriptions and specifications, it is possible to construct an essential model of the system, including a context diagram, an event list, leveled data-flow diagrams and state transition diagrams [You89]. Process specifications may be gleaned from the requirements discussed previously.

1. Context Diagram

It is necessary for the controller to interact with many varied components, as evident from the system block diagram. The context diagram in Figure II-2, shows the context in which the controller must operate. The circle in the center represents the controller. Entities diagrammed outside the circle represent systems outside of the controller's realm of direct control, even though the controller communicates and

dictates timing constraints with them. The buffer, datalink, and track log, diagrammed with a line above and below, represent data storage requirements. The arrows represent the transfer of data between systems.
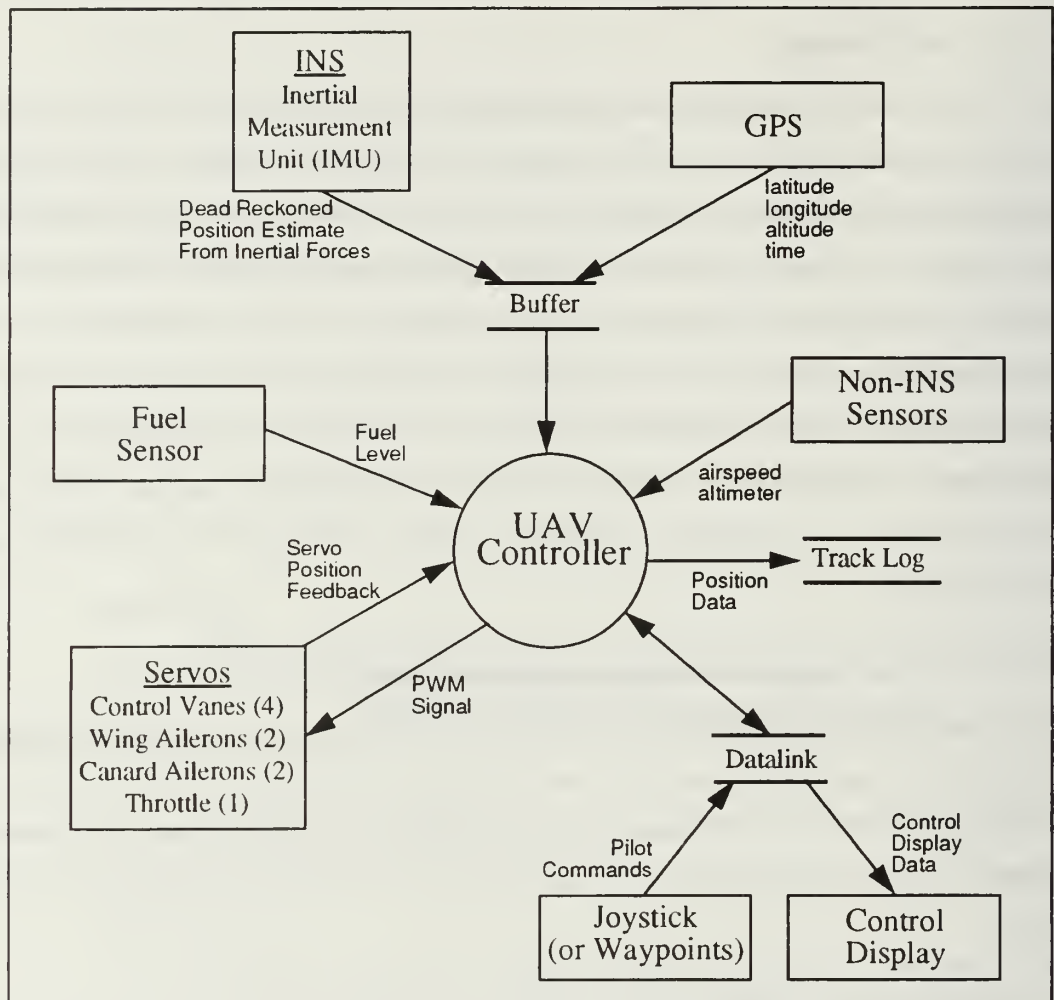
Figure II-2: Context Diagram for UAV Controller

2. Data-Flow Diagrams

Data-flow diagrams (DFDs) are graphical representations depicting the system as a network of functional processes and manifesting the interactions of data flowing between those processes. Although just one of many modeling tools, DFDs are commonly used for operational systems in which the functions of the system are of paramount importance and more complex than the data that the system manipulates [You89]. The top level DFD is followed by sub-level DFDs that further break down the functionality of the top-level processes.

a. Top Level

The top level data-flow diagram (DFD) for the UAV shows the interaction of all of the major processes. As shown in Figure II-3, the controller responds to pilot commands or waypoints, determining control actions and providing feedback to the pilot's control display, which completes the control loop.
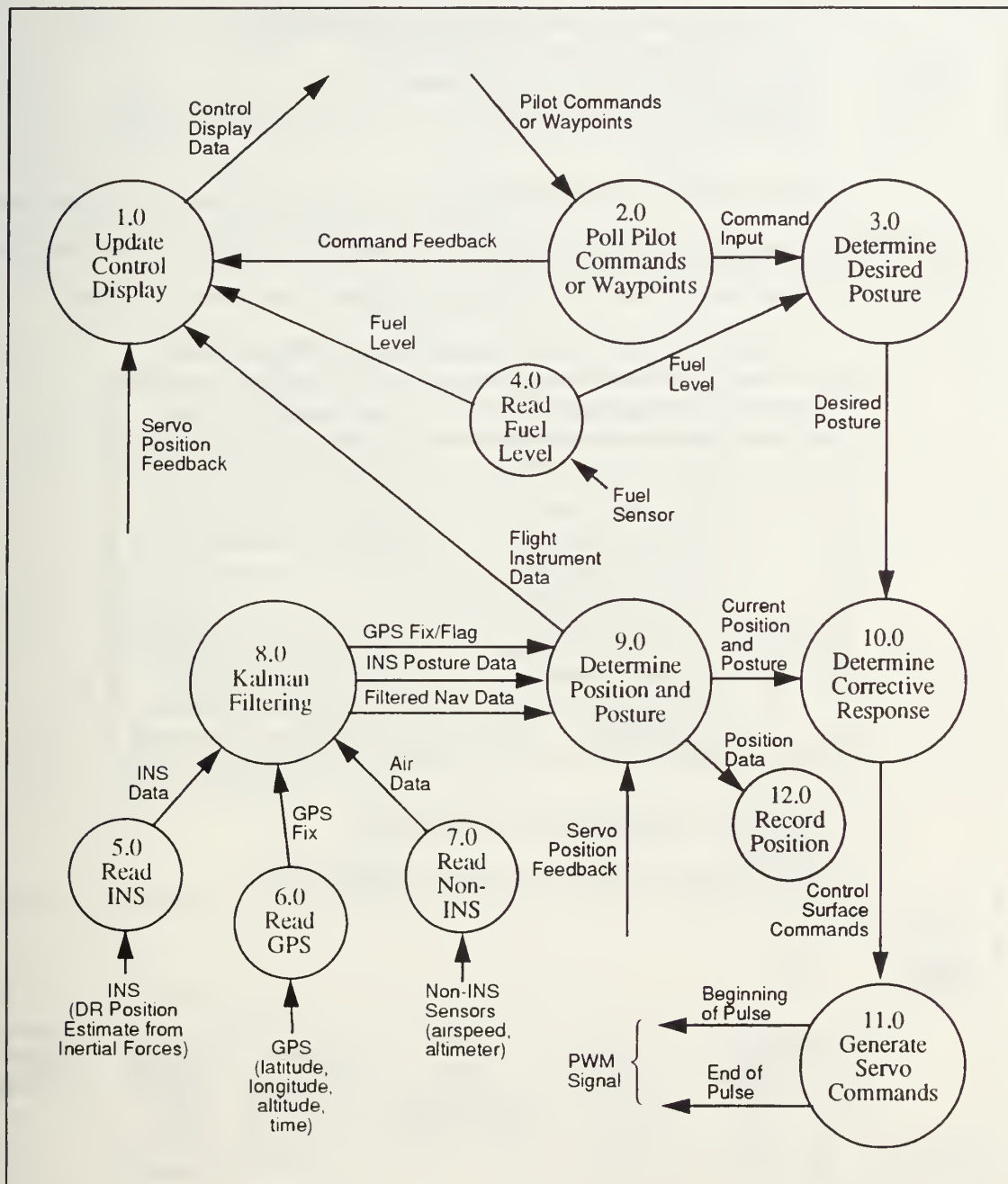
Figure II-3: Top Level Data-Flow Diagram for UAV Controller

b. Single-Process Sub-Level Data-Flow Diagrams.

Many processes in the top level DFD have only one process in their sub-level which just explains the function of the top-level process in more detail. For example, the following processes simply read data from a buffer or A/D port and pass it on without processing:

4.1 = Read digital representation of fuel level from A/D port.
5.1 = Read digital INS sensor data from buffer.
6.1 = Read digital GPS fix data from buffer.
7.1 = Read digital representation of Non-INS sensor data from A/D port.
12.1 = Record position in Track Log.

Three other sub-level DFDs can be described by one process; however, that this process is slightly different depending on the mode of flight. The UAV can be flown directly by a pilot using joystick control or autonomously following a pre-established list of latitude/longitude/altitude waypoints.

TABLE II-1: Process Comparison by Flight Mode

Process	Direct Control Mode	Autonomous Flight Mode
2.1	Read joystick position	Retrieve last/next waypoints
3.1	Convert composite command into vectored commands for each of 3 reference axes (X,Y,Z)	Calculate trackline of acceptable positions between the two given waypoints.
10.1	Call Linear Quadratic routine (Assumption 1) to determine corrective control surface positions	Calculate corrective trackline and, (10.2) call Linear Quadratic routine to determine correct control surface positions.

c. Multi-Process Sub-Level Data-Flow Diagrams

Lastly, the multi-process sub-level DFDs are shown below:
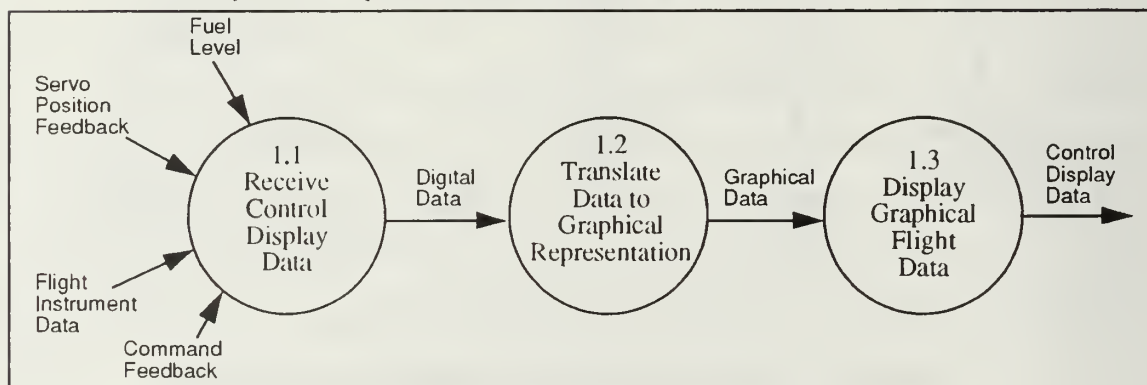
Figure II-4: Process 1.0, Update Control Display

14
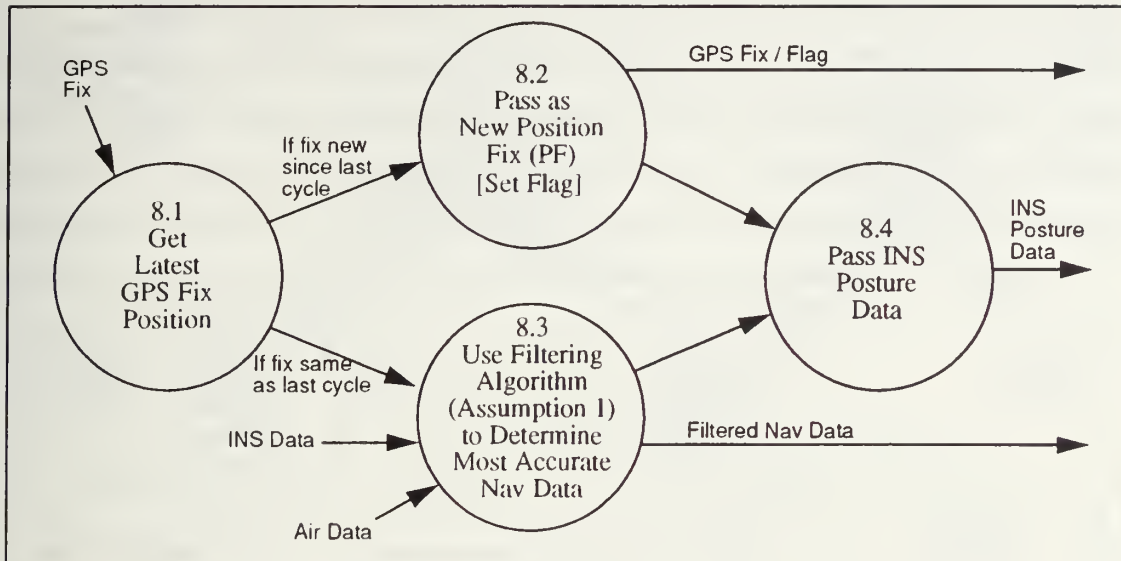
Figure II-5: Process 8.0, Kalman Filtering
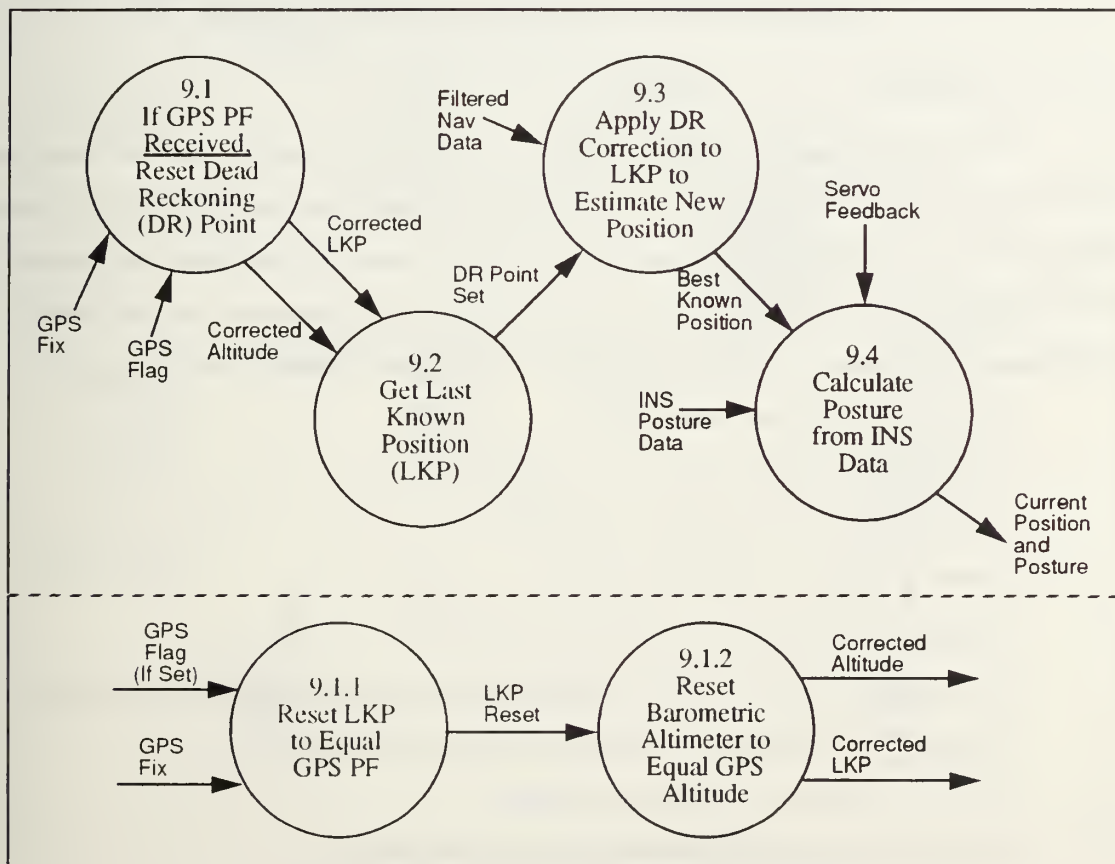
Figure II-6: Process 9.0, Determine Position and Posture

Multi-Process Sub-Level Data-Flow Diagrams (con't):
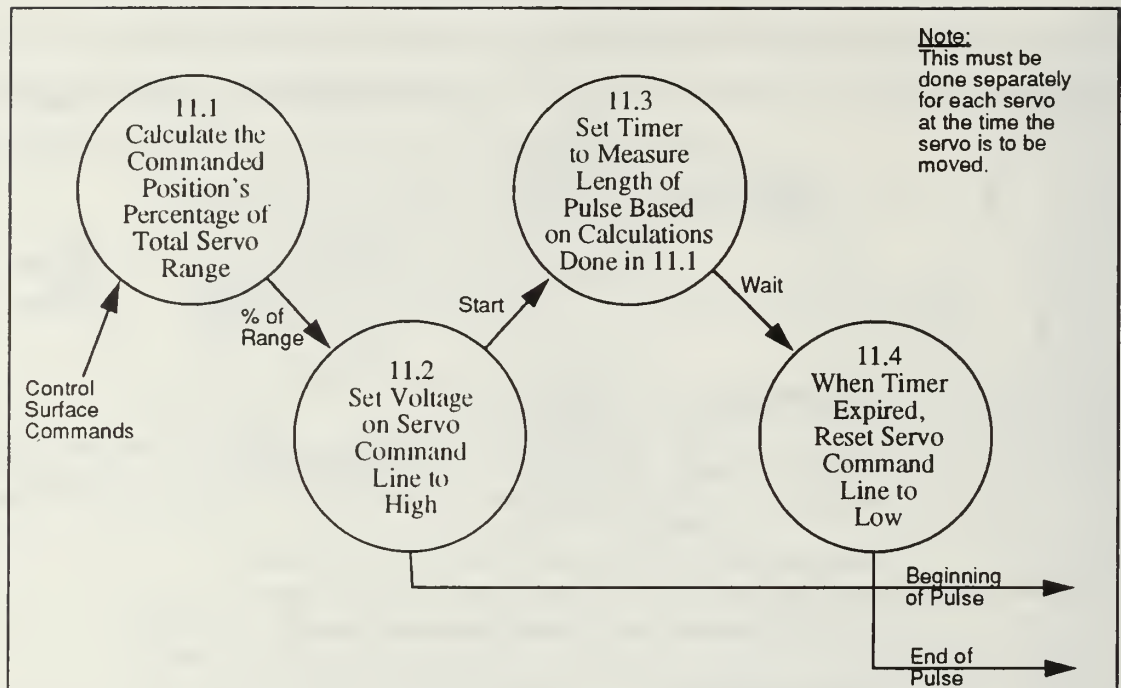
Figure II-7: Process 11.0, Generate Servo Commands

3.   **Event List**

The event list looks deceptively simple.   All events are temporal with the exception of asynchronous joystick inputs from the pilot, but since the command inputs are polled, this event also becomes temporal. Each event is either part of the control cycle or runs on its own frequency which would be an even multiple of control cycles. The challenge is to keep cycles of different durations from becoming out of synch and infringing on each other's resources. The controller must handle the following event parameters:

Pilot inputs new joystick command.

Pilot commands must be polled once per control cycle.

INS data needs to be read once per control cycle.

GPS input needs to be read once per second.

Non-INS data needs to be read once per control cycle.

Each of eight control surface servo command pulses must be generated each control cycle.

The throttle servo command pulse must be generated once per second.

The fuel level needs to be read every 60 seconds.

The control display must be updated at least twice per second.

4.   State Chart

After considering many possible formats to depict state transition information, state charts appeared to best represent the organization of the UAV controller. Developed by D.H. Harel, et al [Har90], state charts combine the state transitions of standard state transition diagrams with process depth typically represented in Warnier-Orr diagrams and then add elements of orthogonality and communication. Orthogonality, represented by a dashed line, indicates separate tasks, and communication, represented by arrows, is a method for allowing different orthogonal processes to react to the same event [Lap92].
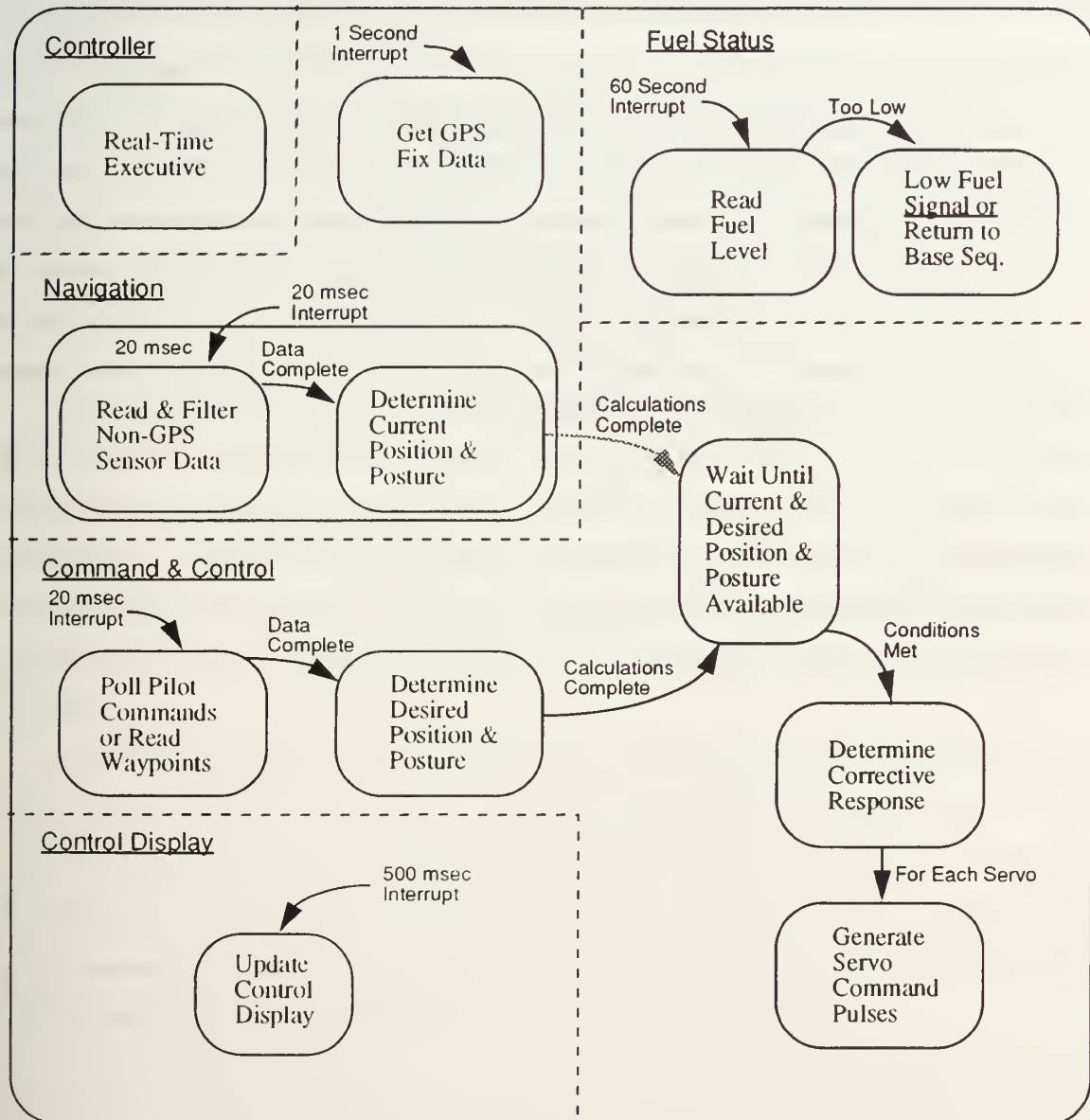
Figure II-8: UAV Controller State Chart

D.   CHAPTER SUMMARY

As an unmanned vehicle, the UAV developed at the Naval Postgraduate School is completely dependent upon its automated systems to provide control of the aircraft in flight. Directing the execution of these automated systems is a controller running a Real-Time Executive program. The correct operation and real-time coordination of all functions on board the aircraft depends on their interaction with this controller.

At present, the UAV is designed to have an on board GPS receiver, an inertial measurement device, and other in-flight sensors. Appropriate data must be selected from this navigation suite, filtered, and analyzed to determine the current state of the aircraft at any given time. This state may have to be converted into a different referential coordinate system. Processing this state via appropriate control algorithms will yield corrective positions for the available control surfaces and the throttle. These controls are moved by pulse-modulated servos, which require a pulse of a specified width be generated and output at a precise time. Pilot commands must be received and the aircraft state may be transmitted through a communication link, using appropriate protocols. Data acquisition, processing, and I/O must be repeated at various intervals and coordinated through a Real-Time Executive (RTE) software program. This RTE is driven by a periodic interrupt and must be robust. The main challenge is to coordinate the complex scheduling of these executed functions to yield smooth and positive control of the UAV.

III. HARDWARE

Hardware selection represents a challenging task. Choosing from the myriad of possible systems and configurations, each with their own cost, advantages, and disadvantages, it is easy to underestimate the ultimate significance of that selection. Indeed, the hardware selected determines most, if not all, of the system's limitations. It impacts the flexibility of the system and the programmer's control over the system. It determines the method by which things can be accomplished on or by the system.

For the UAV controller, the selection of hardware was largely made prior to this research. In general, it was constrained by the following criteria:

- The manufacturer should be from the United States, local if possible.
- All hardware should come from the same manufacturer, for interoperability purposes.
- Therefore, the manufacturer should offer hardware to meet all requirements.
- The composite hardware solution should be modular, for ease of upgrading and maintenance.
- The hardware must be immediately available (not under development).
- The hardware could operate on a standard +/- 5V and +/- 12V power supply.
- The composite hardware solution would fit within the space available on the UAV.
- The composite hardware solution would have minimal weight.
- The hardware could execute commercially available software, including a C compiler.

The reasoning behind choosing an American manufacturer was that if there were any hardware related problems, an American manufacturer, especially one with an office in the local area, would be easiest to contact and could provide the quickest response to requests for technical assistance. A specific operating system was not originally selected, but defaulted to MS-DOS because of the low cost and wide availability of compatible hardware and software. The choice of MS-DOS precludes the use of multi-tasking scheduling strategies, but it was determined that multi-tasking would not be required, at least for the first iteration of the controller.

Ultimately, all controller hardware was purchased from American Advantech, Inc. Their offices and technical staff are located in Sunnyvale, CA. This chapter delineates the most prominent features of the hardware selected. It lists selected specifications for the chosen hardware and discusses the configuration of that hardware as designed for the controller applications. These configurations have been carefully selected to get maximum performance and complete interoperability out of each system component. Numbers listed are Advantech model numbers. Technical data sheets are given in Appendix C.

A.   SYSTEM OVERVIEW

Figure III-1 shows the configuration and connectivity of the overall hardware system. The oval shape represents components that are needed for development only and are detached prior to installation. The remaining components must be mounted on the UAV, most inside the control pod designed by Moran [Mor93]. Racetrack shapes represent sensor and navigation subsystems, most designed through the research of other students and incorporated into this controller design. Rectangular shapes indicate circuit cards or power supplies, most from Advantech, that comprise the central part of the controller. The specifications and configuration of each component is described in this chapter in sufficient detail such that a new user would be able to recreate and understand the present hardware system.
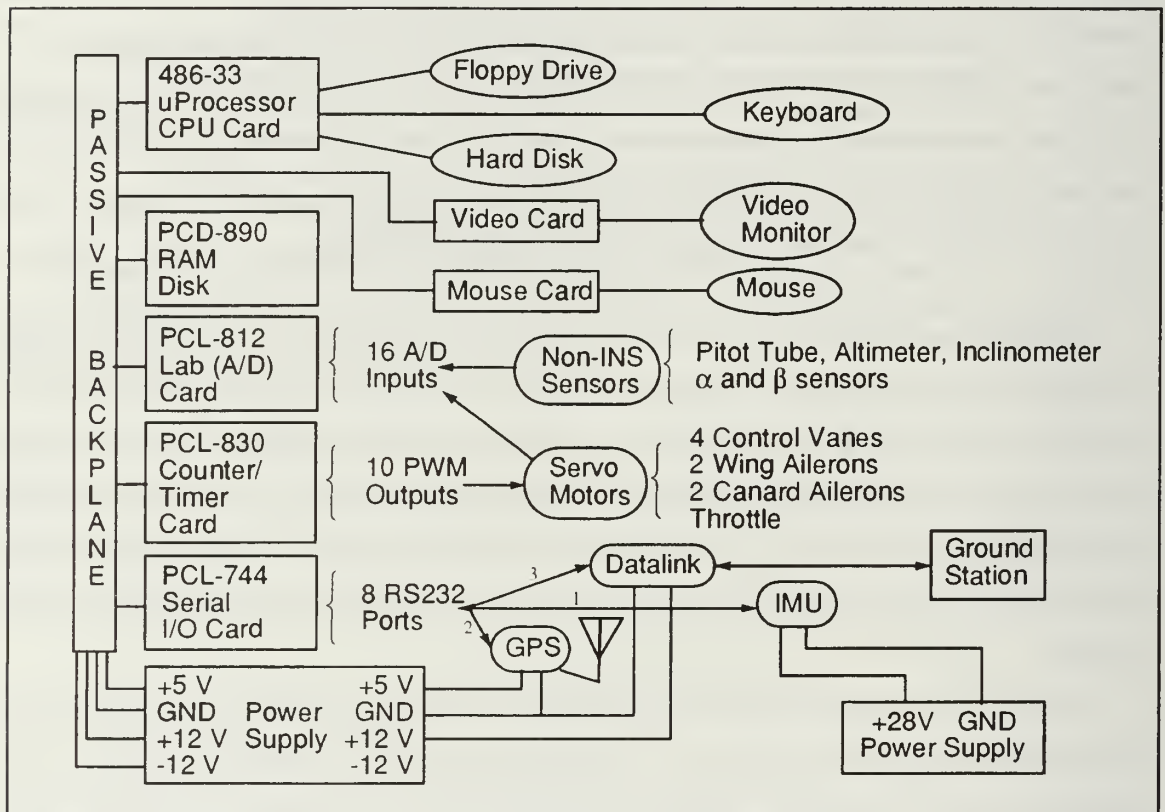
Figure III-1: Overall Hardware System Configuration

B.   PCA-6108: PASSIVE BACKPLANE

This is primarily an external bus, facilitating communication and data transfer between all other hardware components. The other computer circuit boards plug directly into each of the eight, PC/AT compatible expansion slots. This card has a heavy duty, standard block connector for the power supply, making +5V, +12V, -12V, and system ground available to all cards.

C.   PCA-6146: CPU CARD

1.   Specifications

This board contains the central processor, an Intel 80486 running at 33 MHz with 8Kbytes of cache on-chip, 256 Kbytes of 25ns double cache memory and 16 Mbytes of DRAM. Also included on the board are ancillary electronics to support processor functions, such as the Peripheral Interrupt Controller (PIC), Universal Asynchronous Receiver and Transmitter (UART) and counter/timer chips. The board interfaces with the backplane through a 32 bit ISA bus operating at 8 MHz. Control circuits on the board support two floppy disk drives, two IDE hard disks, two RS-232 nine-pin serial ports, one 25-pin parallel port, and a keyboard port. Significant for this research, it features a 4-bit (15 level) interrupt vector and a programmable watchdog timer. The watchdog timer ensures that the CPU will be reset if a program cannot be executed normally, which is useful in real-time systems where a program or power glitch could lock up the system. The maximum power requirement for this board is approximately 2.5 A at +5 V.

2.   Configuration

The CPU card has been configured to support parallel port LPT2, serial ports COM1 on the upper port and COM2 on the lower port, and the floppy disk controller. To accomplish this, the J1 jumper pins must be set as shown in Table III-1. Next, although the controller functions without a hard disk, one is needed for software storage during the initial programming. Thus, jumper JP17 must be enabled (open). The watchdog timer is enabled by closing jumper JP22 and leaving JP23 and JP24 open. The timer interval is set by closing either JP19, JP20, or JP21. Due to the nature of this application, a timeout of 1.5 seconds was selected. Should a power drop, software bug, or infinite loop halt the system, the aircraft would be without positive control for a maximum of 14 seconds, including a total rebooting time of 12 seconds.

When connecting the CPU card to an external panel display, attach the lead wire for hard disk indicator light (usually red and black) to the *HD* connection next to the red LED at the top of the CPU card. Attach the lead wire for the turbo light (usually yellow and black) to JP5. The lead wires from the reset switch (usually red and black) connect to JP4, and the keylock connection is made at made at JP3. Pins 3 and 5 of

JP3 are ground connections, so the black wire should be at the bottom of the connector. Incidentally, pin 1 of JP3 is LED power, and pin 4 controls the keyboard lock.

TABLE III-1: CPU Card Jumper Settings

Jumper	Setting	Jumper	Setting
JP1/1	Close 1-2	JP14	Close 2-3*
JP1/2	Close 1-2	JP15	Open*
JP1/3	Close 1-2	JP16	Closed*
JP1/4	Close 1-2	JP17	Open
JP1/5	Close 1-2	JP18	Open*
JP1/6	Close 1-2	JP19	Closed
JP1/7	Close 1-2	JP20	Open
JP7	Close 2-3*	JP21	Open
JP8	Close 2-3*	JP22	Closed
JP11	Close 2-3*	JP23	Open
JP12	Close 2-3*	JP24	Open
JP13	Close 2-3*		

* Denotes factory setting

3. **Basic Input Output System (BIOS)**

For the system to work properly, the actual hardware and memory configuration must match the setup configured in the non-volatile BIOS chip. This is a CMOS memory chip by American Megatrends, Inc. which stores the system configuration and is read by the processor each time the system reboots. To access the BIOS data, press the DEL key during the initial stages of the bootstrap process, while the processor is checking the available memory. The program will present a main menu from which the user may choose CHIPSET setup, standard CMOS setup, or advanced CMOS setup. For this research, no changes are necessary to the *CHIPSET setup*; all factory default values are used.

In the standard CMOS setup, values entered must correspond to the actual hardware in use. For this research, a 102 MB hard disk and a 1.44 MB, 3.5 inch floppy drive are used. Accordingly, hard drive C is set to *USER TYPE 47*. This should correspond to the following data fields:

TABLE III-2: Hard Drive C Parameters

Cyln	Head	WPcom	LZone	Sect	Size
1024	12	1025	1025	17	102

When booting from the PCD-890 RAM Disk, no actual hard disk is used, so hard drive C must be changed to *Not Installed*. Hard drive D should always be set to *Not Installed*, floppy drive A should be set to *1.44*, and floppy drive B may be set appropriately, if one exists.

Among the many parameters set in the advanced CMOS setup, a few are important. The HD Data Area should be set to *0:300*. This should not be changed unless a different hard disk is used or if the present hard disk is reinitialized. The System Boot Seq should be set to *C: A:*, which forces the system to boot off the hard disk, if one exists. Finally, all cache should be enabled, all video ROM shadow-RAM should be enabled, all adapter ROM shadow-RAM should be disabled, and the system ROM shadow-RAM should be enabled.

D.   PCD-890: RAM Disk

1.   Specifications

The PCD-890 is a solid-state disk emulator with a capacity up to 12 MB, using EPROM, SRAM, or Flash memory chips. For this research, 24 Sony 581000P 128 KB SRAM chips were used to create a 2.88 MB emulated disk. Replacing mechanical drives with the RAM Disk not only allows data retrieval to be accomplished five times faster, but the RAM Disk is also much less susceptible to damage from motion or vibration. Two PCD-890's may be installed on each system, and each PCD-890 has two memory banks that may be configured as either one or two virtual disks. Each virtual disk can be configured as drive A, B, C, or D which are fully software compatible with mechanical drives with no additional software development. Each board features a 3.6 V rechargeable lithium battery that keeps the SRAM charged and lasts up to six months. Memory loss is possible if this battery is allowed to run down; however, there is a connection at CN1 for an optional external battery power. Each card requires only 16 KB of system memory.

2. Configuration

The PCD-890 comes with a utility program which is used to load the on board BIOS chip with the present configuration. This configuration is dependent on the position of various dip switches and jumpers. Jumpers JP1 and JP5 select the type of chips installed in each bank. Both of these should close the connection between pins 2 and 3 to denote SRAM. JP10 and JP11 set the size of the installed chips. These should also close pins 2 and 3 to denote 128 K or larger. Because there is only one PCD-890 installed, the JP9 jumper should close pins 1 and 2. To enable the SRAM battery, close pins1 and 2 on JP4. JP8 sets the interval of the watchdog timer, which is not used on this card.

Dip switches SW1 and SW2 are used to enable each bank and set its drive designation. Position 1 and 2 set the designation. For normal operation, bank 1 must be enabled, unprotected, and set to drive A so that the computer will boot from the RAM Disk in the absence of a hard disk (Recall this bootstrap order was established in the BIOS of the CPU card.). Since it is desired to have one contiguous memory space, bank 2 must be disabled and called something other than drive A. Table III-3 shows the switch settings used for this configuration.

TABLE III-3: Switch Settings for PCD-890

SW1				SW2			
1	2	3	4	1	2	3	4
On for A Off for C	On	On	On	On	Off	On	Off

During system development, it is often desirable to use both the RAM Disk along with a hard drive C and a floppy drive A. In this case, designate the RAM Disk as drive C by turning off switch 1 of SW1. If the PCD-890 is internally designated the same logical name as a hard disk existing in the same system, MS-DOS will automatically assign the PCD-890 to the next available DOS drive, in this case drive D. Note that the utility program will continue to refer to each bank according to the jumper setting, not its DOS drive designation. Also, the BIOS on the CPU card must be updated to correctly reflect the presence or absence of the actual hard drive.

Finally, SW3 sets the memory and I/O addresses assigned to this card. These have been carefully selected to avoid conflicts with other hardware and system services. If two PCD-890's are installed, they

must be set to occupy the same memory address (positions 1, 2, and 3 are the same on both cards), but different I/O addresses (positions 4, 5, and 6 cannot all be the same). SW3 should be set as follows:

TABLE III-4: Switch Settings for PCD-890 SW3

1	2	3	4	5	6
Off	On	Off	On	On	Off

The utility program can be invoked by executing the file named *890* in the PCD-890 directory. Once the switches and jumpers have been properly set, the utility program should mirror that configuration. Drive A is listed as *512KB SRAM* with a disk size of *2.88 MB*. All other entries should read *Not Installed*. Pressing *ESC* will exit the program and load the configuration into the BIOS chip.

Once properly configured and plugged into the backplane, the PCD-890 will automatically install itself in memory during the booting sequence. The only indication will be a message similar to Figure III-2 flashed on the screen for less than one second in between the RAM check sequence and the execution of the AUTOEXEC.bat file. To view this screen, press the *Pause* key to halt the bootstrap process.

```
PCD-890 RAM/ROM DISK BIOS  Rev. B1  (c) Copyright Advantech Co., Ltd.  1992
Configuration:                                              I/O MEMORY


Drive A:  2.88M RAM Disk formatted  (write protect OFF)  0240  D400
BATTERY IS GOOD
```

Figure III-2: PCL-890 Installation Confirmation Message

E.   PCL-744: SERIAL I/O CARD

1.   Specifications

The PCL-744 is an intelligent serial data communications interface card. It provides eight asynchronous, full-duplex RS-232 or RS-422 ports per card, and up to four PCL-744 cards may be used concurrently. It is equipped with a V20 (8088 compatible) 8 MHz CPU, which relieves the central processor of all data handling and I/O flow control tasks. Transmit and receive queues are stored in a 64 KB dual-port RAM buffer, which frees main memory and prevents data loss. *Dual-port* refers to the fact that data can be accessed by both the central processor and the on board CPU. This memory-mapped data transfer is generally much faster than standard memory I/O with data copying. Each card maps to only 8 KB of system memory.

The PCL-744 has a single DB64 female port which connects to a special "octopus" cable branching out into eight DB25 male connectors. Each of the eight ports features complete modem flow control signals (RTS, CTS, DSR, DTR, and DCD) and operate at a programmable communication rate ranging from 50 to 38,400 bps. The PCL-744 uses four 2681 DUART (Dual Universal Asynchronous Receiver and Transmitter) devices, with each 2681 controlling two ports. These two ports share one baud rate divider, which is clocked by a 3.6864 MHz crystal. Baud rates may be selected for each of the two channels independently, but because of the shared divider, both rates must be from the same group:

TABLE III-5: Baud Rate Groups (bps)

Group 1	1200, 2400, 7200, 9600, 38400
Group 2	1200, 1800, 2000, 4800, 9600, 19200

The PCL-744 selects its IRQ level automatically in software and requires a maximum of 1.5 A at 5 V, 120 mA at 12 V, and 170 mA at -12 V, the latter necessary for RS-232 signalling.

2. Configuration

The PCL-744 has no jumpers or switches. Configuration options such as the number of cards, IRQ channels, port numbers, and memory buffer starting addresses are all selected using the setup program. To run the setup program, execute the program named *SETUP* in the PCLS-802 directory. Choose the 744 intelligent card choice to get to the PCL-744 setup screen. On this screen, set the select IRQ number to *10* hex and the select dual port bank to *AUTO*. The start port should be set to *03*. This is because the two COM ports on the CPU card become ports 1 and 2 by default. Pressing *page-down* gives access to the port configuration menu. The *Group Edit* function ensures that all ports are configured identically. Ports 3 through 10 correspond with octopus cable connectors 1 through 8, and are configured as shown in Table III-6.

To install the PCL-744 driver, execute the file *744-DRV.exe* in the PCLS-802 directory. If it installs correctly, the following message appears:

```
PCL-ComLib Communications Driver (Ver 3.00)
PCL-744 Multiport Card 1:  No [05477]  Bank [C800]  Port [03-10]  IRQ 10

Device Driver Setup O.K.
```

Figure III-3: PCL-744 Installation Confirmation Message

Executing the file STD-DRV.exe will also enable control of COM1 and COM2. Both of these executions are done automatically from the AUTOEXEC.bat program during system bootstrap.

TABLE III-6: PCL-744 Serial Port Settings

Ext RxD Buf Size	2K
Baud Rate	9600
Character Length	8
Stop Bits	1
Parity	None
DTR Output State	Off
RTS Output State	Off
CTS Flow Control	No
RTS Flow Control	No
Tx XON/OFF Cntrl	No
Rx XON/OFF Cntrl	No

F.   PCL-812PG: ENHANCED MULTI-LAB CARD

1.   Specifications

The PCL-812 is a high speed, multi-function data acquisition card used primarily in this project to accomplish analog to digital (A/D) data conversions. It features:

- 16 single-ended analog input channels
- Switch selectable bipolar analog input voltage ranges
- A programmable Intel 8253-5 timer to provide internal pacer (trigger) pulsing
- Choice of internal or external reference voltages
- A PCLD-780 wiring terminal breakout board for ease of connection
- Callable software drivers for all card features
- TTL compatible I/O signal levels

Single-ended analog inputs require only one signal wire for each channel. The voltage is measured with respect to common (system) ground. A signal source measured with respect to a reference other than common ground is a floating source. For these signals, a second input called analog ground (A.GND) is available.

The PCL-812 uses an industrial standard 12 bit successive approximation converter (HADC574Z) to convert analog inputs. Typical A/D conversion time is 25 usec. Because an 8 bit register cannot accommodate all 12 data bits, the A/D data is stored in two registers located at the base address +4 and +5.

The least significant bits are in positions 0 (AD0) through 7 (AD7) of BASE +4, and the most significant bits are in positions 0 (AD8) through 3 (AD11) of BASE +5, with AD11 being the most significant. Other important I/O addresses are shown in Table III-7. The PCL-812 requires 16 consecutive bytes of address space and typically draws 500 mA at +5V, 50mA at +12V, and 14mA at -12V.

TABLE III-7: PCL-812 I/O Address Map

Location	Read Function	Write Function
Base Address	Counter 0	Counter 0
Base + 1	Counter 1	Counter 1
Base + 2	Counter 2	Counter 2
Base + 3	Not Used	Counter Control
Base + 4	A/D Low Byte	Ch 1 D/A Low Byte
Base + 5	A/D High Byte	Ch 1 D/A High Byte
Base + 6	D/I Low Byte	Ch 2 D/A Low Byte
Base + 7	D/I High Byte	Ch 2 D/A High Byte
Base + 8	Not Used	Clear Interrupt Request
Base + 9	Not Used	Voltage Gain Control
Base +10	Not Used	Mux Control
Base +11	Not Used	Mode Control
Base +12	Not Used	Software A/D Trigger
Base +13	Not Used	D/O Low Byte
Base +14	Not Used	D/O High Byte
Base +15	Not Used	Not Used

2. **Configuration**

The base address for the PCL-812 is selected using the first six switches of SW1, located at the top of the circuit board. These should be set as shown in Table III-8, giving an I/O address of Hex 220.

TABLE III-8: Switch Settings for PCL-812 SW1

1	2	3	4	5	6	7	8
On	On	On	Off	On	On	On	On

28

Switches 7 and 8 of SW1 control the number of wait states added to the PCL-812 to achieve stable data transfer. It can be configured with zero, two, four, or six wait state delays for each transfer of data. Both switches turned on selects zero delay. Jumpers are used to select the remaining configuration options. Table III-9 shows the present settings and their function.

TABLE III-9: PCL-812 Jumper Settings

Jumper	Setting	Function
JP1	Close 1-2	Use internal A/D conversion trigger
JP2	Close 1-2	Use internal 2 MHz clock for counter channel 0
JP3	Close 1-2	Use internal voltage (JP8) for D/A reference on Ch 1
JP4	Close 1-2	Use internal voltage (JP8) for D/A reference on Ch 2
JP5	Close contact 5	Select IRQ5 to signal A/D completion
JP6	Close contact X	Select no DMA data transfer (DRQ Channel)
JP7	Close contact X	Select no DMA data transfer (DACK Channel)
JP8	Close 2-3	Use -5V for internal D/A reference voltage
JP9	Close 2-3	Select +/- 5V for maximum A/D conversion range

If JP9 is set to +/- 5V, the analog input ranges available for A/D conversion are +/- 5V, +/- 2.5V, +/- 1.25V, +/- 0.625V, or +/- 0.3125V, dependent on a software gain code parameter. These ranges could be doubled by setting JP9 to +/- 10V, but only if Vcc of the system power supply is strictly greater than 12V, otherwise A/D conversions will not be correct. The output of the present power supply is only 11.8V.

Analog connections are made through connection ports CN1 and CN2 on the slot edge of the card. Figure III-4 shows the pin alignment for each connector. For this research, a PCLD-780 wiring terminal breakout board was used to connect signal wires to the ports through ribbon cables.

```
            CN1                            CN2
A/D 0   | 1    2 |  A.GND      A/D 10  | 1    2 |  A.GND
A/D 1   | 3    4 |  A.GND      A/D 11  | 3    4 |  A.GND
A/D 2   | 5    6 |  A.GND      A/D 12  | 5    6 |  A.GND
A/D 3   | 7    8 |  A.GND      A/D 13  | 7    8 |  A.GND
A/D 4   | 9   10 |  A.GND      A/D 14  | 9   10 |  A.GND
A/D 5   | 11  12 |  A.GND      A/D 15  | 11  12 |  A.GND
A/D 6   | 13  14 |  A.GND      D/A  1  | 13  14 |  A.GND
A/D 7   | 15  16 |  A.GND      D/A  2  | 15  16 |  A.GND
A/D 8   | 17  18 |  A.GND      V.REF 1 | 17  18 |  A.GND
A/D 9   | 19  20 |  A.GND      V.REF 2 | 19  20 |  A.GND
```

Figure III-4: PCL-812 Connection Port Pin Alignments

Prior to using the Lab Card, it is necessary to install the PCL-812 driver by executing the file *PCL 812.exe* in the PCL-812 directory. The computer will confirm correct installation with the message, "PCL-812 Driver Version 1.0 is now installed." This execution is also done automatically from the AUTOEXEC.bat program during the system bootstrap process.

3. Calibration

For accurate results, the A/D inputs must be properly calibrated. Five variable resistors (VRs) on the PCL-812 allow accurate adjustment. VR3 and VR5 are used for A/D adjustment, VR1 and VR2 are used for D/A adjustment, and VR4 adjusts the programmable amplifier offset. Executing the calibration program in the PCL-812 directory, the user must specify the input voltage range setting and channel number. Then the program will guide the setting of the programmable amplifier offset, the A/D offset, and the A/D gain. It is important to note that the calibration on one A/D range may cause a small offset on other ranges, so it is suggested to calibrate the A/D range for which the best accuracy is required.

G. PCL-830: COUNTER/TIMER CARD

1. Specifications

The PCL-830 is a multi-function counter-timer and digital I/O card used primarily in this project to generate high-resolution, programmable-duty-cycle square waves used to drive the Pulse Width Modulation (PWM) servos, which move the aircraft's throttle and control surfaces. It provides ten independent 16 bit up/down counters, a 1 MHz crystal oscillator time base, and 16 bit TTL/DTL compatible input and output ports. In the heart of the PCL-830 are two Advanced Micro Devices AMD9513 System Timing Controller (STC) chips used for all counting and timing functions. These STC chips are highly versatile and adaptable to many real time applications, including

- Retriggerable digital timing functions
- Time of day clocking
- Coincidence alarms
- Complex pulse generation
- Frequency shift keying
- Event count accumulation

The STC is addressed by the main processor through two I/O ports: a Control port and a Data port. The Control port provides direct access to the Status and Command registers, as well as allowing the user to update the Data Pointer register. The Data port is used to provide the data used to communicate with all other addressable internal locations. The Data Pointer register controls the Data port addressing. Among

the registers accessible through the Data port are the Master Mode register and five Counter Mode registers, one for each counter. The Master Mode register controls the programmable options that are not controlled by the Counter Mode registers. Each of the five general purpose counters is 16 bits long and is independently controlled by its Counter Mode register. Through this register, the user can software select one of 16 sources as the counter input, a variety of gating and repetition modes, up or down counting in binary or binary coded decimal (BCD), and active-high or active-low input and output polarities. Associated with each counter are a Load register and a Hold register, both accessible through the Data port. The Load register is used to automatically reload the counter to any predefined value, thus controlling the effective count period. The Hold register is used to save count values without disturbing the count process.

The PCL-830 requires 6 consecutive bytes of address space, as follows:

TABLE III-10: PCL-830 I/O Address Map

Location	Read Function	Write Function
Base Address	9513 Chip 1 Data In	9513 Chip 1 Data Out
Base + 1	9513 Chip 1 Command Register	9513 Chip 1 Status Register
Base + 2	9513 Chip 1 Data In	9513 Chip 1 Data Out
Base + 3	9513 Chip 1 Command Register	9513 Chip 1 Status Register
Base + 4	Digital Output Bits 0 - 7	Digital Input Bits 0 - 7
Base + 5	Digital Output Bits 8 - 15	Digital Output Bits 8 - 15

All ports are 8 bits (one byte) wide. When loading data that is longer than 8 bits -- the digital data used to generate PWM signals for this project are 12 bits long -- the low byte must be loaded first, followed immediately by the high byte.

2. Configuration

The base address for the PCL-830 is selected using the first six switches of SW1, located at the top of the circuit board. These should be set as shown in Table III-11, giving an I/O address of Hex 210.

TABLE III-11: Switch Settings for PCL-830 SW1

1	2	3	4	5	6	7	8
On	On	On	On	Off	On	On	On

Switches 7 and 8 of SW1 control the number of wait states added to the PCL-830. It can be configured with zero, two, four, or six wait state delays for each transfer of data. Both switches should be turned on to select zero delay. There is only one jumper on the PCL-830. This jumper (JP1) should close contact 3 to select IRQ3 as the interrupt level. The interrupt is not used in the present configuration, but to use this interrupt, set the Interrupt Enable (CN1 pin 18) low. The positive edge on the Interrupt Input (CNI pin 19) will then generate an IRQ level 3.

Signal connections are made through one of four 20 pin male connection ports. CN1 and CN2, found on the slot edge of the card, are used to interface with the AMD9513 STC chips 1 and 2 respectively. Figure III-5 shows the pin alignment for these connectors. For this research, a PCLD-780 wiring terminal breakout board was used to connect signal wires to the ports through ribbon cables. The servos are attached by first connecting the red and black voltage reference wires to +5 V and GND. The white command wires are then connected to OUT 1 through OUT 10.

	CN1				CN2		
FOUT 1	1	2	GATE 1	FOUT 2	1	2	GATE 6
GATE 2	3	4	GATE 3	GATE 7	3	4	GATE 8
GATE 4	5	6	GATE 5	GATE 9	5	6	GATE 10
SRC 1	7	8	SRC 2	SRC 6	7	8	SRC 7
SRC 3	9	10	SRC 4	SRC 8	9	10	SRC 9
SRC 5	11	12	OUT 1	SRC 10	11	12	OUT 6
OUT 2	13	14	OUT 3	OUT 7	13	14	OUT 8
OUT 4	15	16	OUT 5	OUT 9	15	16	OUT 10
GND	17	18	INT ENB	GND	17	18	Not Used
+5 V	19	20	INT	+5 V	19	20	Not Used

Figure III-5: PCL-830 Connection Port Pin Alignments

It is not necessary to install any drivers prior to using the PCL-830; however, a counter on the AMD9513 must be armed by sending an *ARM* command to the Control port before counting can commence. Once armed, the counting process may be further enabled or disabled using the hardware gating options. Additional commands are provided to step an individual counter by one count, set and clear an output toggle, issue a software reset, clear and set special bits in the Master Mode register, and load the Data Pointer register.

H.   Global Positioning System (GPS) Receiver

1.   Specifications

The Motorola GPS receiver used in this research, model PVT-6, is fully detailed by Twite [Twi94]. It is a fully automatic position finding system that determines and digitally transmits autonomous position, altitude, velocity, heading, satellite tracking status, and correct time in three different, user

selectable formats: Motorola Proprietary Binary Format, National Marine Electronics Association (NMEA-0183) Format, or LORAN Emulation Format. Each of the six parallel channels can find, track, and monitor one NAVSTAR satellite. If three satellites with adequate signal strength and bearing spread are available, a two-dimensional (latitude and longitude) fix is calculated. If four or more usable satellites are available, altitude can also be determined. Instantaneous speed and heading is determined by measuring signal doppler shifts, although without differential corrections, this information is prone to small errors.

2. Configuration

The GPS receiver module and its antenna are fully self-contained units that require no special configuration. The antenna plugs into the coaxial connector on the receiver module. Power and serial data connections are made through a ten pin connector on the back of the unit. A special data cable has been manufactured to provide +5 V and GND to pins 2 and 3 respectively. Serial data communications use pins 8 through 10 and terminate in a DB9 female connector. This is, in turn, connected to PCL-744 octopus cable number 2.

I. INERTIAL MEASUREMENT UNIT (IMU)

1. Specifications

The IMU selected for this project was manufactured by Watson Industries in Eau Claire, WI. Model IMU-600D uses vibrating element sensors to provide the following nine sensor readings:

TABLE III-12: IMU Data Output

Sensor	Scale Limits
X-Axis Acceleration	+3g to -3g
Y-Axis Acceleration	+3g to -3g
Z-Axis Acceleration	+3g to -3g
X-Axis Angular Velocity	+100 to -100 Degrees/Second
Y-Axis Angular Velocity	+100 to -100 Degrees/Second
Z-Axis Angular Velocity	+100 to -100 Degrees/Second
Magnetic Heading	N=7fff, E=c000, S=0000, W=3fff
Bank Angle	+60 to -60 degrees
Pitch Angle	+60 to -60 degrees

Each analog sensor reading is processed through a 16 bit A/D converter and the resulting digital representation of the signal is in two's complement format. To use acceleration as an example, 3g = Hex 7fff, 0g = Hex 0000, and -3g = Hex 8000. In all, there are nine 2-byte words of sensor data. Each word of data is sent as a set of four ASCII characters (0-9 or ABCDEF) corresponding to the hexadecimal representation of the 16 bit word. This complete bank of data is terminated by a carriage-return and line feed, bringing the total size of one data reading cycle to 38 bytes. The sensor data is sent continuously at 9600 baud with one start bit, one stop bit, and no parity bit. At this speed, ignoring any overhead for data formation, a full data bank could be received every 31.7 msec, or just over 31.5 Hz.

The IMU can also receive data. The receive line is used for calibration, so care should be taken to send only the following signals:.

TABLE III-13: IMU Input Signals

Signal	Definition
I	Continuously send bank 1 data (Data described above)
R	Continuously send bank 2 data
Q	Exit Initialization
W	Re-enter Initialization

The IMU normally requires 43 minutes to warm-up and initialize. During initialization, the unit should not be moved for best accuracy. The IMU will send out the bank 1 data stream as it is initializing.

2. Configuration

The IMU is a fully self-contained unit with only one nine pin port for power and serial data connections. Table III-14 shows the pin configuration. A special data cable has been manufactured to provide GND and

TABLE III-14: IMU Pinout

Pin	Function
1	Power GND
2	+28 VDC
3	Signal GND
5	Signal Receive
9	Signal Send

+28 V power to pins 1 and 2 respectively and terminates with a DB9 male connector. This is, in turn, connected to PCL-744 octopus cable number 1. Although the manufacturer attests that the unit can operate with as little as 22 VDC supplied, it must be with respect to system ground for the serial communications to have the proper signal levels. Because of this the +12V to -12V spread available from the system power supply cannot be used; a separate power supply was used for test purposes. Maximum power consumption is 250 mA at +28 V.

J.   DATALINKS

Two different datalinks have been developed for the UAV so far. Both were commercial off-the-shelf (COTS) products that had to meet several criteria:

- Cost limitations
- Weight limitations
- Power limitations
- Size limitations
- Standard serial interface
- Ability to transmit/receive beyond the line-of-sight
- Frequency agility
- Hardware reliability
- Adequate data throughput

The first datalink solution was a X.25 packet radio terminal node controller (TNC) connected to a 19.2 Kbps modem in combination with a UHF wide-band transceiver developed by Reichert [Rei93]. This is a robust system that meets or exceeds almost every criteria. Reichert's estimate of required data throughput is accurate in scope, but may change slightly in the final design. For example, he lists the control refresh rate as 40 Hz while this controller operates at 32 Hz. He estimates 8 bits per servo update, while the present configuration uses 12; however, the present configuration could be reduced to 8 bits per servo with no noticeable change in performance. An updated throughput requirement estimate is shown in Figure III-6. Using this updated requirement, the capacity of this datalink, which yields 19.2 Kbps simplex or 9600 bps duplex, is exceeded. Possible solutions would be to reduce the servo input to 8 bits and to refrain from downlinking INS and non-INS sensor data for every control cycle.

The second datalink solution was a direct-sequence spread spectrum UHF datalink as developed by Bess [Bes94]. This datalink also used a modified X.25 protocol with a top transfer speed of 19.2 Kbps. It has programmable length packets and performs its own error correction and flow control. Spread spectrum transmission has the added advantages of being less susceptible to jamming signals and may be operated

without an FCC license. The unit includes a serial data cable that terminates with a DB9 female connector. This is, in turn, connected to PCL-744 octopus cable number 3.

Controls to be Uplinked

Device	Qty	Refresh Rate	Bits per Update	Required Throughput
Throttle	1	16 Hz	12	192 bps
Control Vanes	4	32 Hz	12	1536 bps
Wing Ailerons	2	32 Hz	12	768 bps
Canard Ailerons	2	32 Hz	12	768 bps

Servo Positions to be Downlinked

Device	Qty	Refresh Rate	Bits per Update	Required Throughput
Throttle	1	16 Hz	12	192 bps
Control Vanes	4	32 Hz	12	1536 bps
Wing Ailerons	2	32 Hz	12	768 bps
Canard Ailerons	2	32 Hz	12	768 bps

Navigation and Sensor Data to be Downlinked

Device	Qty	Refresh Rate	Bits per Update	Required Throughput
GPS Receiver	1	1 Hz	544	544 bps
INS Sensors	1	32 Hz	304	9728 bps
Non-INS Sensors	2	32 Hz	12	1152 bps

Total Datalink Throughput Requirement

Device	Required Throughput
Controls	3264 bps
Servo Feedback	3264 bps
Sensor Data	11424 bps
Estimated Datalink Overhead (15%)	2693 bps
Total Datalink Throughput Required	**20645 bps**

Figure III-6: Datalink Throughput Requirements for Ground Control

The first datalink was not used in this research because of its complexity and licensing requirements. The second datalink did not perform well in the laboratory environment, occasionally locking up or dropping

out of service for no apparent reason. Although configured for 9600 bps throughput, actual results were less than half of that. It also required cycling the power and manual configuration to be restarted. Once autonomous flight is achieved, the datalink will diminish in its importance, but as long as direct ground control is required, the datalink is the lifeline and the weakest link in controller communications.

K.   ANCILLIARY EQUIPMENT

In addition to the aforementioned hardware integral to the controller itself, ancillary equipment was necessary to form the complete system. In order to control the UAV, servos and a source of power are needed. Any COTS model airplane servo motors could be used, as described by Stoney [Sto93], provided they generate sufficient torque to hold the control vanes position in the thrust stream. The servos used for this UAV were Futaba high torque model FP-S34. The red and black wires connect to +5 V system power and system GND respectively. The white command signal wire is attached to the PWM output signal from the PCL-830 counter/timer card. The white feedback wire connects to one of the A/D analog input channels on the PCL-812 lab card. Power for this research project was generated by a standard AMAX 200 W power supply that provided up to 20 A at +5 V, 8 A at +12 V, and 0.5 A at both -5 and -12 volts, which was ample for all hardware used. Power generation and storage will vary according to the UAV design. Although not investigated in this research, it is a very technical problem that affects the operation of all hardware.

A video graphics adapter (VGA) card linked with a standard VGA monitor and a Microsoft in-port mouse, together with the hard disk and floppy drives represented the detachable part of this system hardware which was necessary for system development only. Once the controller software was developed, it was transferred to the RAM Disk. The system was then configured to boot from the RAM Disk and automatically invoke the controller program. The keyboard and monitor are also detached prior to installation into the airframe; however, the software must be modified slightly to accept user commands from the datalink prior to operation without the keyboard and monitor.

L.   CHAPTER SUMMARY

This chapter gives a detailed description of the specifications of the hardware selected for the UAV controller, and the configuration details necessary to enable all equipment to interact together, including those subsystems developed by other students. Using this information, system users and follow-on researchers will know and understand how the system was designed to operate and the configuration details necessary to reconstruct a similar system. For additional information about the specifications or operation of this hardware, see Appendix C or the published user's manuals.

IV. SOFTWARE

Once the hardware has been selected, it is the software that actually shapes the operation of the controller. The hardware and software have a symbiotic relationship by which neither can function without the other. The software must operate within the confines of the hardware's capabilities and configuration, and the hardware fulfills its tasks in a coordinated manner by taking its direction from the software. Software gives the system its function and its personality. It provides for the interface by which the user comes to know and recognize the system, it provides the transfer and organization of all the data crucial to the controller's operation, and it coordinates the functions of and sets the cadence for the hardware components. Where hardware is the body, software is the life blood.

This chapter will provide an overview of the scope of the software written for this research. Beginning from the original requirements, it will specify the conventions, definitions, and structures necessary to help the reader understand the code. It will detail the software environment in which the software is designed to operate, and it will briefly describe the function of the various procedures. A complete listing of the code is included in Appendix A.

A. OVERVIEW

The UAV is a multi-faceted project bringing together many varied disciplines. Since the scope of this research was to design the Real-Time Executive (RTE) for a central controller, it required the assimilation of many sub-systems into one interoperable, cohesive, control system. These sub-systems were developed by other students as part of an ongoing development effort. Among the many sub-systems previously developed, this RTE was specifically designed to coordinate datalink development by either Bess [Bes94] or Reichert [Rei93], navigation system development by Twite [Twi94] and Hallberg [Hal94], servo control development by Merz [Mer92] and Moran [Mor93], and aeronautical control algorithm development begun by Davis [Dav92] and Brynestad [Bry92], and continued more recently by Bolyard [Bol94] and Moats [Moa94].

1. Requirements

As the backbone of the UAV controller, the RTE designed for this research acts like the conductor of an orchestra, directing the flow of data and cueing the execution of the various controller functions at the correct moment in time. The controller's most basic requirement is to provide positive control of the aircraft. This includes analyzing the present state, comparing the present state with the desired state, and making the

necessary adjustments. In order to provide this control, the control software must meet several other criteria as well. Specifically:

- The software must be able to receive data from all flight sensors (GPS, INS, non-INS).
- The software must recognize and store a correct and complete data package from each sensor.
- The software must recognize and discard corrupted data packages received from any sensor.
- The software must always maintain the most recent data available from each sensor.
- The software must provide Kalman filtering of navigation data, selecting the most appropriate source for use by the control algorithm
- The software must recognize command input and determine desired aircraft posture and position.
- The software must calculate corrections necessary to correct any deviations from desired posture and position.
- The software must generate PWM signals for servo motors to effect the necessary corrections.
- The software must be able to transmit and receive data through the datalink as necessary.

Other requirements are not as obvious, but became apparent during the development of the system. They include a predisposition to be written in C language and to be interrupt driven. In addition, the RTE must deal effectively with exceptional circumstances, and it must be flexible and clearly written.

a. The RTE should be written in C language

Because of the low level programming requirements, the project did not fit well with an object-oriented paradigm. In addition, aeronautical engineering students developing the control algorithm are using a program called Matrix-X which generates modules in C code. To interoperate with these modules, and for reasons mentioned in Chapter II, C was chosen as the programming language.

b. The RTE should be interrupt driven

As delineated in Chapter II, hard real-time systems are required to meet timing deadlines imposed by the outside environment or risk system failure. When the control loop is initiated by a timed interrupt, it assures that the system will always execute positive control functions at a uniform interval which is easily regulated. This not only frees the processor to do other things when not executing the control loop, but also allows the control loop to interrupt slower (by processor standards) processes, such as generating servo command pulses or writing to the screen, prohibiting their occurrence from affecting system timing deadlines.

c. The RTE must deal effectively with exceptional or emergency occurrences

What if the datalink fails or the buffers are full? What if the CPU gets into an infinite loop or comes to a halt? What if the operator wants to reset the system? What if the GPS or IMU do not deliver

a complete message? These or any number of other possible mishaps or exceptions can occur, and the system must be able to respond appropriately and reestablish positive control of the aircraft. Any problems with system integrity come under the auspices of the RTE.

d.   *The RTE must be flexible to evolve with the rest of the system*

The development of the UAV was designed in five phases, as outlined by Reichert [Rei93]. As the control of the UAV evolves from remote ground control to full automation, the software must be flexible enough to evolve with it. To facilitate this requirement, it should be modular in design; each procedure should be self-contained and should perform a specific function.

e.   *The RTE must be clearly written to facilitate follow on work*

Just as this is not the first research project on the UAV, it will not be the last; however, interoperability and cohesiveness will still be requirements. The software programs are intended to be self-explanatory through form, logic, and inserted comments. Where they are not, this research document is intended to serve as a programmer's manual for all functions of the software.

2.   Definitions

All preprocessor directives, including compiler token definitions, global variable definitions, and procedure prototypes are contained in the lone header file called *DEFS.h*. An index of all other variable names is contained in Appendix B, which may be used as a glossary by subsequent programmers. Special complex variables are stored in C structures, as described below.

a.   *Structures*

Very few structures are used in the control software, as the necessary data types are not complicated. The first is *struct T_GPS*, introduced by Twite [Twi94] and fully defined in the header file GPSTRUCT.h. This structure is globally maintained and gives access to all possible information obtained from the GPS receiver by simple structure-member reference. For example, the control algorithm could access the degrees of latitude of the present position by simply using the variable name gps.pcs.latitude.degrees.

The second is a *PANDL*, which represents a pointer and its length. This is the preferred method of passing data contained in buffers of differing length. It allows one structure argument to be passed and yet allow the same procedure to handle the various length buffers consistently.

3. Conventions

As a means of standardization, a set of conventions have been established in the design of the control software. Any procedures not part of the RTE, but subsequently added to the controller software (hereafter termed *participating procedures*) should conform to these conventions. In order to avoid contention and interference, the RTE must maintain control over several critical parameters, including execution timing, data transfer, memory allocation, and the operation of the hardware, particularly the datalink.

a. The RTE must maintain control over all timing

This is the defined function of the RTE, yet for it to be effective, participating procedures should be of relatively constant execution time. Recursion and loops must be used carefully, and the participating procedure should not call another procedure that should be under the control of the RTE. The challenge of programming the RTE is then reduced to a complex scheduling problem among a relatively small number of processes which all have concise scope and operating parameters.

b. The RTE must maintain control over all data

As a corollary to the previous requirement, the RTE also maintains control over all aspects of data storage and transfer. This includes I/O port number definitions, flow control definitions, and actual I/O requests including input from the keyboard, output to the screen, or data transfers with the datalink. This is crucial to maintain coordinated operation of all controller functions. Any participating procedures must refrain from making their own data transfer calls, unless it is the express function of that procedure. Data information placed in a buffer is passed using the PANDL structure defined above.

c. The RTE must maintain control over memory allocation

In the course of operation, a given procedure may be called many times by the RTE. Memory allocation is a relatively slow procedure, and should be minimized and carefully managed. Any procedures that allocate memory must clear that memory prior to return to the calling program. It is preferred to have the RTE allocate the memory and then pass that PANDL to the participating procedure fill the buffer and modify the length.

d. The RTE must maintain control over the hardware

It is the function of the RTE to direct the execution of the hardware. Unless it is their defined purpose, participating procedures should not send signals to hardware or in any manner change the operating

41

parameters of the hardware established by the RTE. This will preclude the RTE from coming into contention with the operation of one of the participating procedures.

e. *Messages coming from the ground must have a set format*

Because the datalink was only used to transmit information down to the ground in this research, this format has not been completely established. The read_datalink() procedure is written to expect a special character to denote the start of the message (presently using '#'), followed by a two byte integer representing the length of the message in bytes, followed by the message itself. It is possible to also include a one byte action code after the message length to help the RTE determine what action to take with the incoming message. Participating procedures that uplink information to the RTE must follow this convention for the message to be properly deciphered.

B.   COMPILER CONFIGURATION

The software was developed under Borland C/C++, version 2.0. Invoking this program using the command BC, without any flags, brings the user into an integrated development environment (IDE). The IDE, otherwise known as the Programmer's Platform, includes a multi-file editor, multiple overlapping windows, an integrated debugger, a built in assembler, and support for in-line assembly of other object modules. Pull down menu selections are at the top of the screen, and most are similar to other graphical user interfaces. The following compiler configuration parameters are important to ensure that the code will compile properly.

1.   Project File

Using the *Project* pull down menu gives access to the project file. The project files are kept in the C:\borlandc\bin directory and perform two important functions. First, the status of the screen (or desktop) and all preferences selected, including compiler options, are stored in the project file. Then, when the project is opened, the screen and all preferences are automatically returned to the settings selected for that project. Second, the project contains a list of files to be compiled at run time. This allows the user to specify other files, like header files or separate object modules, to be included in the compilation. As shown in Figure IV-1, two such files are required for correct compilation of the controller software. 812CL.lib is an object code library for the intrinsic functions used to operate the PCL-812 board. MOXA-CL.obj is an object code module for the intrinsic functions used to access the PCL-744 board. According to the manufacturer, both

boards require that the intrinsic functions be used to access the boards. Experience has confirmed that the intrinsic functions are also the easiest and most efficient method of accessing the functions of these boards.

Project: Monitor				
File Name	Location	Line	Code	Data
Monitor.c	..\..\Control	946	14788	4209
812CL.lib	..\..\PCL-812\C	n/a	n/a	n/a
MOXA-CL.obj	..\..\PCLS-802\LIB\C	n/a	n/a	n/a

Figure IV-1: Compiler Project Screen

2. Compiler Options

The *Options* pull-down menu gives access to the selected compiler options. Most of these may be set to the user's preference, but several are important and should not be changed. Under *Code Generation*, the large memory model should be selected and *Automatic Far Data* should be checked. Because of the "segment:offset" addressing scheme in the computer, several memory models are available. For each item of code or data, the compiler can either generate explicit segment and offset addresses or can use the offset alone within a default segment address. The large model generates explicit segment and offset addresses for all data items, thus allowing an unlimited amount of code and data with only one constraint: no single data item can exceed 64 Kbytes [Bar89]. This model is shown graphically in Figure IV-2, and is necessary for direct memory access and for setting far pointers used in the interrupt service routines (ISRs).

In the *Entry/Exit Code Generation* menu, neither *Standard Stack Frame* or *Test Stack Overflow* should be checked. Because of the heavy use of the stack for ISRs and console functions, like printing to the screen, the stack frame should be as large as possible. Additionally, with the *Standard Stack Frame* option turned off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return codes. This makes the resulting code shorter and faster. The *Test Stack Overflow* generates code to check for stack overflow at run time. This code is not necessary, and can cause run-time problems in the controller. Similarly, under the *Linker* option, no stack warning should be checked. During interrupts, the stack is not where the stack checker expects it to be. Under *Optimization* options, select optimize for speed. Under normal conditions, the compiler will choose to optimize for size, choosing the smallest code sequence possible. With this item toggled, the compiler will choose the fastest sequence for each task. This is important since the program does not come close to exceeding the 2.88 Mbytes available on the RAM Disk,

43

but is significantly time-constrained. Last, under *Directories*, the compiler is operating with the *Include* directory set to C:\borlandc\include, the *Library* directory set to C:\borlandc\lib, and the *Output* directory set to C:\control.
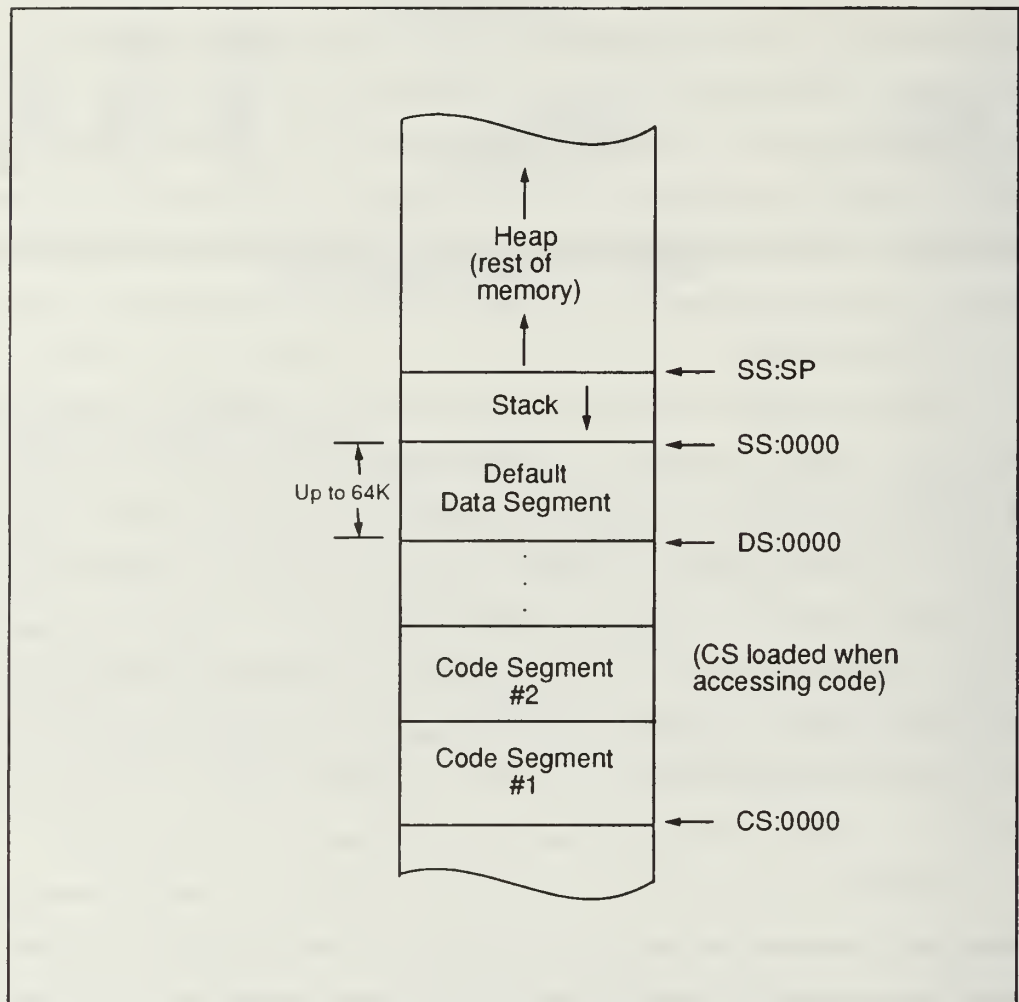
Figure IV-2: Diagram of Large Memory Model

C.   SYSTEM INITIALIZATION

This section highlights initializations that must be completed before the program can start the flight management unit (FMU) sequence to control the UAV. For proper operation, the software program must have been compiled in accordance with the compiler options described above. Then, once the program is invoked, the main() procedure is the first code to execute.

1. Software Initialization

In the beginning of the main() procedure, special exit handling routines are set up. The atexit() procedure directs the program to execute the shut_down() procedure whenever the program is terminating from any reason. This is a handy function because the termination point can come anywhere, yet the shut_down() procedure will always be executed, assuring that the ISR vectors have been returned to normal, the allocated memory has been freed, and the I/O ports have been properly closed.

The ctrlbrk() procedure establishes a return point in the program in the event of a control-break key sequence. It can be seen in the break_handler() routine that this is a method for completely restarting the program without having to reboot. After the exit handling routines are set up, the global structures needed to hold sensor data are allocated. This includes a struct T_GPS, PANDLs for the IMU and GPS information, and data and param arrays for the PCL-812.

2. Hardware Initialization

After establishing exception handlers and memory allocation, the main program calls initialize_hw() . This procedure controls the parameters of the hardware that must be set up in software, which is necessary for three of the hardware cards: the PCL-812 Lab Card, the PCL-744 Serial I/O Card, and the PCL-830 Timer/Counter Card.

a. PCL-812

The PCL-812 relies solely on the param array for information concerning its operation. It is important that the hardware and software configurations match. Specifically, param[4], IRQ level, must match the setting of jumper JP4; param[7], trigger level, must match the setting of jumper JP1; and param[17], gain code, must match the setting of jumper JP9. Other important parameters are param[5] and param[6], the product of which divides the 2 MHz clock to determine the speed of the internal trigger, and param[14], [15], and [16] that set how many A/D conversions will be done and on which analog inputs. With the param array established, the initialize_hw() procedure calls PCL-812 function 3 to initialize the hardware and PCL-812 function 4 to begin A/D conversions.

b. PCL-744

The PCL-744 uses the same library of software functions as the PCLS-802 Serial I/O Card, which is not used in this project. The PCLS-802 software has three major parts: first is the complete RS-232 based software device driver for I/O processing and control; next are the interface libraries which allow the

use of high-level programming languages to control serial communications; and third are application programs which allow troubleshooting of the serial communications. All of these interface library procedures begin with "sio_" and so are hereafter termed *sio functions*. Advantech engineers have confirmed that the use of these library functions is the only method available for accessing the PCL-744. These sio functions are fully described in Chapter 3 of the PCLS-802 PC-ComLIB Manual by Advantech. The initialize_hw() procedure uses these sio functions to configure each of the eight serial ports as follows:

- 9600 baud rate
- 8 data bits
- 1 stop bit
- No parity
- DTR off
- RTS off
- Hardware flow control off
- Software flow control off

Compiler variables for the bit configurations needed for these settings are found in HEAD-C.h. Companion sio functions are used in check_hardware() to read the values set for each port. After configuring the ports, initialize_hw() opens each port, flushes the receive and transmit buffers, and sends initialization codes to the GPS receiver and the IMU.

c. PCL-830

The last section of initialize_hw() initializes the PCL-830 card. The original software was written by Moran [Mor93] for another circuit card that also used AMD9513 System Timing Controller (STC) chips, and so it could be ported over with minor modifications. This code is well documented by Merz [Mer92]. Each timer is configured as follows:

- Counter clock source set to F1 (1 MHz)
- 8 bit wide data bus (mandatory for the PCL-830)
- Binary counting on falling edge, counting down repetitively
- Reload counter from Load or Hold register
- Disabled data pointer increment (this is controlled by for-loops in software)
- No gating control
- Output control set to toggle on terminal count

This configuration is equivalent to a specialized version of the AMD9513 Mode F. Under this configuration, the individual counter is alternatively loaded from its Load and Hold registers. First, the counter loads the value from its Hold register and puts the output high (5 V) upon terminal count (counting down to zero). Then the counter loads the value from its Load register and toggles the output low (0 V) upon terminal count. The value in the Load register creates the desired length of the PWM pulse, which should be between 0.6 ms and

2.4 ms. The sum of the Load and Hold registers sets the PWM signal refresh rate, which should not exceed 10 ms [Dav92]. Notably, this mode differs from Mode C in that the counters are not required to be loaded and armed manually, except initially. This initial arming is done at the end of initialize_hw().

D.   INTERRUPTS

Interrupts can come from two different sources: hardware and software. Both hardware and software interrupts are decoded by hardware chips called Peripheral Interrupt Controllers (PICs), and both use the interrupt vector table to find the location of the interrupt service routine (ISR), a small program designed to address the cause of the interrupt. Hardware interrupts typically call the processor's attention to an external event, such as a key stroke or other asynchronous action. Conversely, software interrupts are like instructions; they are part of, and therefore synchronous with, the running program.

The lowest 1 Kbytes of memory is allocated for an interrupt table that can store the four byte "segment:offset" address for each of 256 ISRs. The correct ISR is located by its number, no matter where it is located in memory. The CPU simply has to multiply the interrupt number by 4 (since each segment has four bytes) and jump to the address it finds at the resulting offset in segment 0. For example, the address of the ISR that serves interrupt 70 is found in segment 0 at an offset of 70 X 4 = 280 = 118h. The interrupt table does not contain the ISR code itself, but the address of the beginning of the ISR code. To change the execution of an ISR, it is only necessary to change the address in the interrupt table for the desired interrupt. Upon occurrence of an interrupt, the CPU will place the value of the program counter and all internal registers on the stack for future reference. Then it will look up and jump to the address of the ISR. Upon completing execution of the ISR, the CPU then retrieves the information it had placed on the stack and resumes normal operation [Nor85].

The PIC is the chip that translates external interrupt request signals (IRQs) into hardware interrupts, allowing external devices to generate interrupts. The microprocessor itself has only two interrupt lines: one for maskable interrupts and one for non-maskable interrupts. Maskable interrupts are those that can be disabled or enabled in software. The PIC assigns priorities to its eight interrupt lines, with line 0 programmed for the highest priority by default. When one of the lines is activated, the PIC blocks all IRQs of equal or lower priority. It continues to block these IRQs until it receives an end-of-interrupt (EOI) code from the processor. The processor communicates with the PIC at the microcode level. When its maskable interrupt line goes high and interrupts are enabled, it queries the PIC which is the highest pending IRQ and then jumps to the associated interrupt vector. The CPU card has two 8259A PICs. The output line of the secondary PIC

is attached to IRQ 2 of the primary PIC, and the output line of the primary PIC is connected to the processor. This allows 16 different IRQ signals to be recognized by the processor. When IRQ 8 or higher is generated, the processor queries the PICs and finds IRQ 0 through IRQ 7 of the secondary PIC is cascaded through IRQ 2 of the primary PIC [Rie93].

1. Generating Software Interrupts

As discussed in Chapter II, the cadence of the entire control loop is built around the periodic occurrence of a software interrupt. Because the CPU will immediately jump to the ISR when interrupted, placing the beginning address of the control loop program in the interrupt table, and generating a periodic interrupt to jump there, will guarantee a consistent frequency of control loop execution. For the UAV, the entity that executes the control loop is called the *Flight Management Unit* (FMU). Within the start_fmu() procedure, the ISR for the RTC interrupt (70h) is replaced with a pointer to the control loop procedure new_vector(). Start_fmu() then proceeds to generate a periodic interrupt, as described below.

Every computer has some version of a *Programmable Interval Timer* (PIT). Intel 80X86 processors usually use a 8253 or 8254 PIT that has three independently programmable 16 bit counters that can be configured in any of six counter modes. On the CPU card, one of these counters is used to periodically refresh the DRAM; one is used to generate tones for the speaker. The third timer is used to generate an interrupt 8 (IRQ 0) at 18.2 Hz used to adjust the current time and date in the system BIOS area. It seemed like a simple process to change the frequency of this interrupt and "hook" it for the FMU; however, this led to problems. Because IRQ0 is the highest priority, all other computer functions, such as serial communications, disk operations, and keyboard activations, were all blocked out by the PICs. In addition, some parts of MS-DOS that require periodic service hook this interrupt, and these functions could be adversely affected by changing the frequency of the interrupt. Most significantly, the engineers at Advantech confirmed that the 8254 functions on the CPU card are part of an "integrated chip set" and could not be accessed independently. For these reasons, another timer had to be used. A timer on the PCL-830 could be used, but this was discounted because it was on another card and would generate unnecessary data traffic on the backplane bus. Fortunately, there is another timer available on the CPU card that generates interrupts -- one that is not widely documented, but is available on the hardware. It is called the Real-Time Clock.

2. The Real-Time Clock

The real-time clock (RTC) is part of the Motorola MC146818A CMOS chip shared by the system BIOS. As compared to the 8254 PIT, it has several disadvantages:

- The RTC is less flexible; it handles only 15 possible interrupt frequencies between 2 Hz and 32767 Hz.
- The default ISR switches off the RTC interrupt after a time-out expires.
- The RTC and IRQ 8 are not well documented. Most of this information was gleaned from online sources from the Internet.

Still, the RTC is perfect for this research application because it avoids all of the problems listed above for the 8254 chip:

- The RTC allows the higher priority keyboard and I/O interrupts to proceed normally.
- The RTC is not polluted with side effects.
- The RTC is available for use on the hardware being used.

Because the RTC exists outside of the normal address space, it cannot contain directly executable code. It is communicated with through I/O ports 70h and 71h. Port 70h is the index register and port 71h is the data register, as defined in DEFS.h. All internal registers of the RTC are accessed by setting an index at port 70h and reading from or writing to port 71h. The output from the RTC is in hexadecimal. Figure IV-3 details the CMOS memory allocation. The ten clock data registers are not used in this research, although it is envisioned that the system clock will be updated from the GPS time data in the future. In order to use the RTC to generate periodic interrupts, only the four status registers are used.

There are a few caveats when programming the RTC. First, the data register must always be read from or written to after writing to the index register. Also, there should not be a long delay between writing to the index register and reading from or writing to the data register. Waiting too long between the two operations can cause a malfunction of the CMOS chip [Dun86]. Interrupts must be disabled while programming the RTC. The non-maskable interrupt (NMI) must also be disabled. Since the chip is non-volatile, it continues to work even in the event of a system reboot caused by a NMI. The system reads vital parameters from the chip, such as memory size and configuration. Malfunction of the RTC chip is to be avoided at all costs. Therefore, it is safest to toggle the NMI off by toggling bit 7 of the index register when selecting the status register to use. The following describes how the status registers are utilized.

49

The first 14 bytes of the MC146818 chip consist of ten read/write data registers and four status registers, two which are read/write and two which are read only.

The format of the 10 data registers is:

00h	Seconds	(BCD 00-59, Hex 00-3B) Note: Bit 7 is read only
01h	Second Alarm	(BCD 00-59, Hex 00-3B)
02h	Minutes	(BCD 00-59, Hex 00-3B)
03h	Minute Alarm	(BCD 00-59, Hex 00-3B)
04h	Hours	(24 Hr. Mode: BCD 00-23, Hex 00-17)
		(12 Hr. AM: BCD 01-12, Hex 01-0C)
		(12 Hr. PM: BCD 81-92, Hex 81-8C)
05h	Hour Alarm	(Same as Hours, above)
06h	Day of Week	(01-07, Sunday = 01)
07h	Date of Month	(BCD 01-31, Hex 01-1F)
08h	Month	(BCD 01-12, Hex 01-1C)
09h	Year	(BCD 00-99, Hex 00-63)

The format of the four status registers is:

0Ah Status Register A (read/write)

Bit 7 (Read Only)	1 = update cycle in progress, data undefined
Bits 6, 5, 4	22 stage divider of 32.768 KHz time base
Bits 3 - 0	Rate selection bits for interrupt

0Bh Status Register B (read/write)

Bit 7	Cycle update: 0 = disabled, 1 = enabled
Bit 6	Periodic interrupt: 0 = disabled, 1 = enabled
Bit 5	Alarm interrupt: 0 = disabled, 1 = enabled
Bit 4	Update-ended interrupt: 0 = disabled, 1 = enabled
Bit 3	Square wave output: 0 = disabled, 1 = enabled
Bit 2	Clock data mode: 0 = BCD, 1 = Binary
Bit 1	24/12 Hour Selection: 0 = 12, 1 = 24
Bit 0	Daylight Savings Time: 0 = disabled, 1 = enabled

0Ch Status Register C (read only)

Bit 7	Interrupt request flag: 1 if any of bits 6 - 4 are 1 and appropriate enables in Reg. B set to 1. Generates IRQ8.
Bit 6	Periodic interrupt flag
Bit 5	Alarm interrupt flag
Bit 4	Update-ended interrupt flag
Bits 3 - 0	Not Used

0Dh Status Register D (read only)

Bit 7	Valid RAM: 0 = dead battery or disconnected, 1 = good
Bits 6 - 0	Not Used

Figure IV-3: Organization of CMOS Memory

Status register A is used to select an interrupt rate. The basic oscillator frequency is 32,768 Hz, set in bits 4 - 6. The lower four bits (0 - 3) of status register A select a divider for this oscillator. The resulting frequency is used to generate an interrupt 70h, or IRQ 8. The system initializes these bits to 0110 binary, which selects a 1,024 Hz frequency according to the following formula:

$$InterruptFrequency = OscillatorFrequency >> (rate - 1)$$

which can be simplified as

$$InterruptFrequency = \frac{65536}{2^{rate}}$$

Table IV-1 lists the subset of interrupt frequencies likely to be used for the UAV controller. These frequencies are generated when the corresponding rate is specified in DEFS.h. Presently, the controller is executing a 32 Hz control cycle. The fastest frequency possible is 8 KHz using a rate of 3. When using a rate of either 2 or 1, the counter rolls over, resulting in the same frequencies as rates 9 and 8 respectively.

TABLE IV-1: CMOS Interrupt Frequencies

Rate	Frequency
10 (0Ah)	64 Hz
11 (0Bh)	32 Hz
12 (0Ch)	16 Hz
13 (0Dh)	8 Hz
14 (0Eh)	4 Hz
15 (0Fh)	2 Hz

Status register B contains a number of flags. To enable the chip to generate periodic interrupts, bit 6 must be set. Status register C is read only and also contains a number of flags. When several interrupts of the RTC are connected to IRQ 8, these flags make it possible to detect which interrupt caused the IRQ 8: periodic interrupt, alarm interrupt, or update ended interrupt. Lastly, the PIC status register must be unmasked. Each PIC has an 8 bit mask that disables selected IRQs. The American Megatrands BIOS disables IRQ 8 at startup. By clearing bit 0 of the secondary (slave) PIC, IRQ 8 is enabled.

These actions generate a periodic interrupt, but only a single one. Unless status register C is read, IRQ 8 will not be generated again. This means status register C is read inside the ISR, even though its content is not important for this application. The PICs also come into play here. Since the PIC blocks all IRQs of equal or lower priority upon the occurrence of an interrupt, the next periodic interrupt cannot be generated

until the PICs receive an *end-of-interrupt* (EOI) code from the processor. This is accomplished by directly outporting an EOI (value of 20h) to I/O addresses 20h and A0h, the addresses of the master and slave PICs respectively. These repetitive actions must be done on every occurrence of the periodic interrupt and so are accomplished inside the ISR in a subroutine called reset_int().

Common practice when writing ISRs is to jump to the old ISR after executing the new one, but because the old ISR halts the periodic interrupt, this method was not used. Without the old ISR, some interrupt 15h BIOS functions will fail; however this did not manifest any problems in the present configuration. If necessary to alleviate this in the future, store at address 0040:009b a double word value that is at least 976 and jump to the old ISR. The default ISR subtracts 976 from the value at that address and halts the RTC periodic interrupt if the result is less than zero. 976 is derived as the number of microseconds that elapse between two invocations of interrupt 70h if the RTC is counting at its default frequency of 1024 Hz. The default ISR also issues an interrupt 4Ah when timed out [Bro92].

In addition to resetting the interrupts, the ISR, new_vector(), also increments a count of the number of cycles and calls the actual control loop procedure, execute_cycle(). It is within this control cycle that all of the I/O and flight control operations takes place.

E.   THE CONTROL CYCLE

The control cycle is embodied in the execute_cycle() procedure. It is called by new_vector() upon each occurrence of the periodic interrupt. During the control cycle, the controller first retrieves the information it needs to determine the state of the aircraft by invoking each of four I/O device drivers; next, it calculates any corrective actions necessary in the flight_control() procedure; and finally, it generates the PWM signals necessary for the servo motors to effect those corrective actions in the cmd_to_servos() procedure. It is important to note that not all of these procedures are called during each cycle. Using modulo division of the cycle count, it is possible to regulate the interval and period of the various functions. The objective is to keep the task load for each control cycle relatively steady. The actual timing of these functions is dependent on the frequency of the interrupt. For example, the IMU is read every fourth cycle for control purposes, but is only downlinked to the ground twice each second. This programming strategy keeps the RTE flexible and the timing parameters easily modified. Each of these functions will be examined in detail.

1.   I/O Device Drivers

Four I/O device drivers exist within execute_cycle() to take care of the data transfer and storage from the four primary sources of data for the FMU. They are read_imu(), read_gps(), read_atod(), and

52

xmit_to_gnd(). Each of these procedures reads one complete data message from their appointed interface and places that message in a pre-established global data buffer.

As described in Chapter III, the complete data message from the IMU is 38 bytes long and terminates with a carriage return. The global buffer, imu_buf->ptr, has 100 bytes allocated in the main program. Because of this buffer restriction, read_imu() reads the length of the queue and truncates it to 100 bytes. It then determines if the data in the receive buffer constitutes a partial or full message. If a partial message exists, it reads it away before reading in the next full message. Last, it confirms whether the message read was a complete 38 byte message and sets a flag accordingly.

Read_gps() works much the same way, except that a full position message is 68 bytes and terminates with a carriage return and a line feed. A 500 byte buffer is allocated in main(). This procedure pares down the receive buffer until the buffer size constitutes at least one full message and at most one full message plus a partial message. If a partial message exists, it reads it away and then reads in the next full message. It also confirms whether the message read was a complete 68 byte message and sets a flag accordingly.

The A/D process on the PCL-812 card was initiated in the initialize_hw() procedure. To read the data generated into the data array, read_atod() needs only to call PCL-812 function 5, and the data is read in automatically. Since the data received is a 12 bit digital conversion scalar value, determining the actual analog voltages requires a calculation similar to that done in the show_air_data() procedure.

Depending on the mission and the mode of flight, varying amounts of data will be required to be transmitted to, and received from, the ground station through the datalink. This data transfer is done by the read_datalink() and xmit_to_gnd() procedures. Read_datalink() was written for functional completeness, but is not utilized in this research. Based on the last convention in Section A.3 of this chapter, read_datalink() will look for the beginning-of-message character, then read the message length, convert the length to an integer, and use the length given to read in the appropriate number of bytes constituting the message. If a message is read, it is stored in the global buffer dl_buf, and a flag is set accordingly.

The xmit_to_gnd() procedure has the advantage that the length of the message to be transmitted is known from the PANDL passed in. The procedure uses the sio_putb function to transfer the data to the datalink's transmit buffer. Experimentation has shown that the sio_write function works equivalently well. The only exception that must be considered is a full transmit buffer. This situation should never occur under normal operating circumstances; however, in the event that it does occur, the buffer is flushed under the

53

assumption that the data presently attempting to be sent is the most recent and therefore more valuable than the old data that was clogging the buffer.

2. Flight Control

Now that the controller has all the information it needs from the various I/O drivers, all that remains is to calculate the control inputs necessary to fly the airplane and move the servo motors appropriately. The flight_control() procedure in this research is only a placeholder for the control algorithm module being developed in the Aeronautical Engineering Department. Eventually, this procedure will provide the Kalman filtering options for choosing the appropriate navigation data from what is available and, using this data, will calculate integer control commands for the throttle and for each of the standard three-dimensional control surfaces: aileron, elevator, and rudder. These control surface commands are then passed to the cmd_to_servos() procedure, which translates the three-dimensional control surface commands into individual vane commands. The integer command expected by cmd_to_servos() presently represents the number of degrees of deflection, but it could be changed to any agreed upon standard between flight_control and cmd_to_servos. Cmd_to_servos() then sends the appropriate signals to the PCL-830 to generate the precise PWM signal needed by each vane servo. This concludes the control loop segment of the program and meets all of the requirements for positive control outlined above. The RTE now returns to the point of execution prior to being interrupted and continues its normal activity until the occurrence of the next periodic interrupt.

F. USER SERVICES

When not executing the control loop, the computer is primarily available for user-oriented services. These services include a gamut of small procedures designed to interact with the system user and provide information. Because the system runs on interrupts, the control loop described above appears to be running in the background, while these user services utilize the screen and keyboard and appear to run in the foreground. The initial and primary interface with the user is the menu() procedure. Menu() presents a command line, prompting for user input. A new user may respond with a question mark, which yields a menu of possible choices, as shown in Figure IV-4. The first three choices, check hardware, start flight management unit, and quit flight management unit, have been described above. The remaining choices are described below.

```
Archytas Monitor Program

Command ('?' for help):  ?

The following are valid commands:
────────────────────────────────

(c)heck hardware
(s)tart flight management unit
(q)uit flight management unit
(f)light data menu
(m)emory contents display
(r)egister contents display
(i)nterrupt vector display
(d)os command
(t)erminate program

Command ('?' for help): __
```

Figure IV-4: Main Menu Screen

Choosing *flight data menu* invokes the show_flight_data() procedure, which presents a secondary menu as shown in Figure IV-5. This menu enables the user to inspect and verify the contents of the global buffers containing the flight data gathered during the control loop from each of the I/O device drivers. This includes ASCII representations of the untranslated GPS message and the output from the IMU, the hexadecimal values and corresponding voltages of all A/D analog sources, and the present positions of all servos. These values represent the instantaneous buffer contents at the moment in time they are retrieved. If the FMU is running, the buffer contents may change immediately after being read.

```
Display which data?
─────────────────

(g)ps position
(i)mu data
(a)ir data
(s)ervo positions

Choice: __
```

Figure IV-5: Flight Data Menu Screen

The *memory contents display, register contents display*, and *interrupt vector display* main menu choices enable the user to inspect the contents of any block of memory, the contents of all storage and segment registers of the processor, and the ISR address stored in the interrupt table for any given interrupt respectively.

Similar to the utility of a debugger, these procedures were used primarily during the development of this program and are included for future debugging needs. When programming at such a low level, interacting with individual memory locations and I/O ports, it is often necessary to have the utility of these procedures.

Lastly, the *dos command* menu choice invokes an MS-DOS shell that the user can work in while the FMU is still running. All basic DOS functions, such as copying files, directory listings, and invoking small programs are available, as long as the intended task does not require BIOS interrupts 81h or 83h. Terminating the program will also yield a DOS prompt, but only after the program completes its shutdown sequence.

One other secondary menu is available to the user, although it is not listed in the main menu. It is invoked by the *Ctrl-Break* or *Ctrl-C* key sequence. Both of these are standard key sequences used when the user wants to terminate what is executing. In this case, the break_handler() procedure is invokes and a menu similar to Figure IV-6 is displayed.

```
Why did you break?
_____

(c)old reboot machine
(w)arm reboot machine
(r)estart program (reinitialize hardware)
(g)o back to main menu
(t)erminate program

Choice: __
```
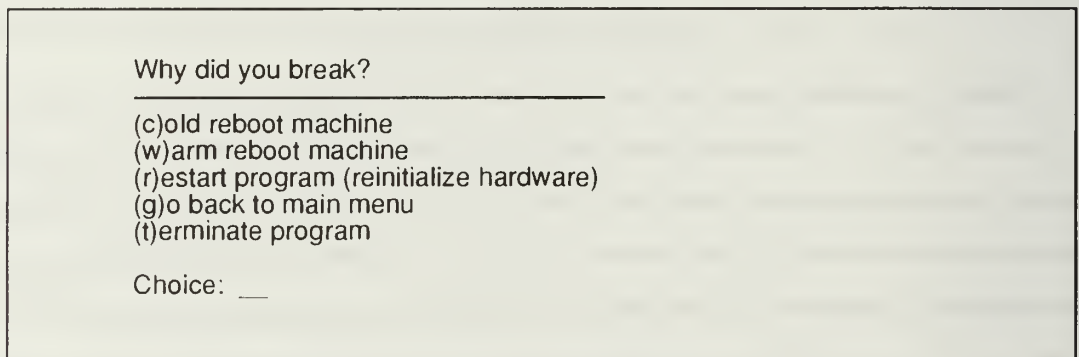
Figure IV-6: Break Handler Menu Screen

The most drastic response to this prompt is a *cold boot*. This is similar to the system initialization done when first powering up the computer, including all diagnostic and memory checking sequences. A *warm boot* is similar to the *Ctrl-Alt-Del* key sequence and causes the computer to reboot without the diagnostic and memory checking sequences. This makes it slightly faster and less disruptive than a cold boot. Both of these options invoke the bootstrap() procedure, passing in the chosen parameter of cold or warm. Because disk caching is used, bootstrap() first flushes the caches to insure that no information is lost and reboots the computer. If the computer is operating correctly, the user may elect to re-initialize only the controller hardware. This *restart program* option causes the computer to execute the hardware shutdown sequence and then start the program again from just after the buffer allocation in main(). Other choices allow the user to return to the main menu or terminate the program.

G.   CHAPTER SUMMARY

This chapter gives a detailed description of the software program written to function as the control software for the UAV.   The reader should understand the full scope of the endeavor, including the requirements and guidelines under which it was written, and the interoperability with the hardware, including those sub-systems developed previously by other students.   This chapter serves as a programmer's manual, to aid the understanding of the code shown in Appendix A, as well as to set conventions and guidelines for subsequent code to follow from other work on the UAV project.   The operation of the Real-Time Clock chip, in particular, is not documented elsewhere, and is therefore completely detailed in this chapter.

V. CONCLUSIONS

The goal of this research was to create a functional central controller for an UAV. This controller is envisioned to integrate various subsystems designed by other students as part of an overall, interdisciplinary development project. It represents the airborne half of a full UAV control system that is planned to evolve from remote ground controlled flight to fully autonomous flight in five phases of development [Rei93].

A.   ACCOMPLISHMENTS

From the general goal to create an UAV controller, specific operational requirements were derived. From these operational requirements, system hardware was selected and design parameters were codified. In the course of the design and synthesis of the controller, several significant milestones were achieved:

- The system hardware was assembled and configured for proper operation.
- A method for generating periodic interrupts was determined and successfully implemented.
- Multi-path serial I/O was achieved using the PCL-744 card.
- Datalink subsystems were successfully integrated.
- Air data and navigation subsystems were successfully integrated.
- Servo control subsystems were successfully integrated.
- The RTE was designed and implemented to interrelate and coordinate all subsystems.
- Communication and programming standards were developed.
- Fault tolerant provisions were made to bolster system reliability.
- User interfaces were designed and implemented.

The UAV controller was designed to be as simple as possible, given the hardware on hand and the anticipated task load. A real-time executive (RTE) program, initiated by timed interrupts at various intervals, calls appropriate task modules, and repeats this process indefinitely. The use of interrupts enabled the processor to keep busy during slow (by processor standards) processes, such as generating servo command pulses. Under this configuration, the challenge of programming the RTE was then reduced to a complex scheduling problem among a relatively small number of processes which all have concise scope, known parameters, and demonstrated characteristics. The only immutable programming requirement was to arrange the process schedule of the RTE such that a called process can complete execution prior to the initiation of another process, and so that the resources of interrupted processes are not needed by the interrupting process.

This research details the inter-relations of the design criteria used for this controller, to give the reader a better understanding of the overall system. From this understanding, present design decisions become apparent, and future development is facilitated. The future development described below is recommended to develop a more effective and efficient controller design.

B. RECOMMENDATIONS

In the course of development, it became evident that several improvements would be necessary for the final implementation of the system. These included standardization of the command structure and improvements in data conversion and transfer, in addition to some general system modifications. Also, other subsystems with which the controller must interact, namely the aeronautical control module, were not completed as of this writing. These areas are recommended for future research and development and are briefly delineated below.

1. Command and Control Structure

The basic operation of the controller is to determine the state of the aircraft, compare that state to a state commanded by the pilot, whether that pilot is a human or the controller executing a set of preprogrammed waypoints. Since the control module has not been completed, there is no standard command syntax in place. Neither is there a structure for communicating these commands to and from the control module. For this research, a temporary procedure named flight_control() was written which simply generated vane commands in degrees. A future control module should have the capability to read the necessary flight data from the global registers, *determine the commanded state*, and generate control vane angles compatible with the cmd_to_servos() routine. It is this middle function that needs to be carefully defined.

The control module is expected to output these commands for each of the standard three-dimensional control surfaces: aileron, elevator, and rudder. It is presently the responsibility of the cmd_to_servos() procedure to translate these commands into appropriate coordinated commands for the eight control vanes planned for installation on the Archytas [Sto93]. This translation is currently incomplete, especially considering that the translation parameters must change as the aircraft transitions from vertical to horizontal flight. This area requires additional study unless this translation process is absorbed into a control module that incorporates both the flight_control() and cmd_to_servos() functions.

2. Data Generation and Conversion

Outside of the control algorithm, the controller's main function is to gather and disseminate necessary data to appropriate functions. The faster this data can be generated, the better the controller can perform. Several factors are impeding optimum performance, as described below.

First, the direct memory access (DMA) form of data transfer should be used where possible. This would preclude the waste of processor resources to perform memory to memory copying of data. Several

accessory boards, specifically the PCL-744 serial card and the PCL-812 lab card advertise a DMA capability. This utility was attempted, but never successfully implemented during this research.

Second, the serial ports can transfer data more quickly. The serial card was set up at 9600 bps to match with other subsystems that had been designed to operate with a standard RS-232 serial port, which normally operate at that speed. The ports of the PCL-744, however, can be configured as high as 38400 bps [PCL-744 Manual, p. 16]. Each port should be optimized separately, since some of the connected subsystems, like the datalink, can be configured to run at variable speeds, while other subsystems, like the GPS and the IMU run only at 9600 bps.

Third, the IMU is too slow. As explained in Chapter III, the fastest possible message frequency would be 31.5 Hz. Empirical data has shown the actual message frequency to be closer to 20 Hz. To have new IMU data for every control cycle requires slowing the cycle or increasing the output rate of the IMU. Watson Industries does offer various options which can increase the speed of the IMU, and these options should be explored.

Fourth, the speed of the datalink is too slow. As shown in Chapter III, the data transfer requirements for remote controlled operation from the ground is above the capacity of the datalink. This will become less of a factor as the UAV development progresses towards autonomous flight, but it will always be exacerbated by increasing the link overhead as propagation quality deteriorates. Field experimentation will show which data is more crucial to control and which data could be sent less frequently. Overall, for positive remote control, no more than 100 msec can elapse between pilot command input and the associated movement of the control surfaces [Kam93]. In the early stages of development, the datalink is the weakest and yet most important link in the control process.

3. General System Modifications

Because the datalink proved to be so unreliable, user menu selections were entered directly from the keyboard. When the keyboard is removed to place the controller in the aircraft, these menu programs must execute through the datalink. Fortunately, because of the case statements used to execute menu choices, the user interface programs require only minor changes to read user input from the datalink, rather than the console keyboard.

Second, the GPS routines need to be fully implemented. The procedure read_gps() was written in place of Twite's procedure Slave_gps(), which was not fully completed. It is Twite's program that decodes the data stream from the GPS and places the navigation in a global structure, as discussed in Chapter IV. This

60

convenient access to GPS data will not be available until Twite's software is completely implemented. This includes the potential for accessing GPS data other than the position change status message by uplinking commands to the GPS receiver through the PANDL *gps->in* [Twi94, p. 125].

Third, the 25 pin serial connectors on the PCL-744 octopus cable are much too heavy and bulky for actual implementation. For the RS-232 connections, only eight wires have the potential of carrying signals and, because flow control is not used, only three wires are actually used. During actual implementation, it is recommended that customized cables be used.

Last, a multi-tasking or multiple processor CPU board should be investigated. Even without the processing-intensive control algorithm, this controller is extremely constrained by real-time deadlines. The addition of other processing requirements could force the system to be run at an unacceptably slow interrupt interval. Although upgrading to a faster CPU would ease the problem somewhat, a multi-tasking or segregated multiple processor environment should produce a better solution with higher flexibility and greater throughput.

C.   SUMMARY

Through this research, the goal of designing and building an UAV controller has been successfully completed. The resulting aggregation of hardware and software represents a functional shell to which improvements can be made, and into which other subsystems, developed in the future, may be added. From the initial primary research question, down to the final working implementation, this research quantifies the system mandates and documents the conceived solutions. This controller represents a proof-of-concept for unmanned control of air vehicles, and one that, with the addition of a suitable control module, is ready to fly.

LIST OF REFERENCES

[Bar89]   Barkakati, Nabajyoti, *The Waite Group's Turbo C Bible*, Howard W. Sams and Co., Carmel, IN, 1989.

[Bes94]   Bess, Philip K., *Spread Spectrum Applications in Unmanned Aerial Vehicles*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1994.

[Bol94]   Bolyard, John W., *Stability and Control Analysis of a Ducted Fan Unmanned Air Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1994.

[Bro92]   Brown, Ralf and Kyle, J., PC Interrupts: A Programmer's Reference to BIOS, DOS, and Third-Party Calls, Addison Wesley, 1992.

[Bry92]   Brynestad, Mark E., *Investigation of the Flight Control Requirements of a Half-Scale Ducted Fan Unmanned Aerial Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1992.

[Dav92]   Davis, Joseph P., *The Design of a Robust Autopilot for the Archytas Prototype via Linear Quadratic Synthesis*, Masters Thesis, Naval Postgraduate School, Monterey, CA, December 1992.

[Dun86]   Duncan, Ray, *Advanced MS-DOS*, Microsoft Press, Redmond, WA, 1986.

[Gla83]   Glass, Robert L., *Real-Time Software*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[Hal94]   Hallberg, Eric N., *Design of a GPS Aided Guidance, Navigation, and Control System for Trajectory Control of an Air Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1994.

[Har90]   Harel, Lachover, Naamad, Pnueli, Politi, Sherman, and Trauring. "STATEMATE: A working environment for the development of complex reactive systems." *IEEE Transactions on Software Engineering*, April 1990.

[Hel87]   Heller, Philip, *Real-Time Software Design: A Guide to Microprocessor Systems*, Birkhauser Publishers, 1987.

[Int92]   Intel Corporation, *80486 Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, 1992.

[Kam93]   Kanadine, Tom, UAV Project Engineer, McDonnell Douglas, Inc., St. Louis, MO, Phone Conversation, May 1993.

[Koc92]   Koch, Paul A., *Flight Testing of a Half-Scale Remotely Piloted Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, September 1992.

[Kue93]   Kuechenmeister, David R., *A Non-Linear Simulation for an Autonomous Unmanned Air Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, September 1993.

[Lap92]   Laplante, Phillip A., *Real-Time Systems Design and Analysis: An Engineer's Handbook*, IEEE Press, 1992.

[Lea91]   Leatherman, Brent L., *An Approach to Integration of Real-Time Software for an Autonomous Underwater Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1991.

[Mar93]   Marquis, Carl W., *Integration of Differential GPS and Inertial Navigation Using a Complementary Kalman Filter*, Masters Thesis, Naval Postgraduate School, Monterey, CA, September 1993.

[Mer92]   Merz, Paul V., *Development and Testing of the Digital Control System for the Archytas Unmanned Air Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, December 1992.

[Moa94]   Moats, Michael., *Automation of Hardware-in-the-Loop Testing of Control Systems for Unmanned Air Vehicles*, Masters Thesis, Naval Postgraduate School, Monterey, CA, September 1994.

[Mor93]   Moran, Patrick J., *Control Vane Guidance for a Ducted-Fan Unmanned Air Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1993.

[Nel92]   Nelson, M.L., Brutzman, D.P., Byrnes, R.B., Badr, S.M., *Real-Time Systems*, Term Paper, Naval Postgraduate School, Monterey, CA, February 1992.

[Nor85]   Norton, Peter, *The Peter Norton Programmer's Guide to the IBM PC*, Microsoft Press, Redmond, WA, 1985.

[Rei93]   Reichert, Frederick W., *Datalink Development for the Archytas Vertical Takeoff and Landing Transitional Flight Unmanned Aerial Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1993.

[Rie93]   Riemersma, Thiadmer, 100115.2074@CompuServe.com, Electronic Mail, November 1993.

[Sav85]   Savitzky, Stephen R., *Real-Time Microprocessor Systems*, Van Nostrand Reinhold Company, 1985.

[Sta88]   Stankovic, J.A. and Ramamritham, K., *Tutorial: Hard Real-Time Systems*, Computer Society Press of IEEE, 1988.

[Ste73]   Stein, Jess, *The Random House Dictionary of the English Language*, Random House, Inc., New York, NY, 1973.

[Sto93]   Stoney, Robert B., *Design, Fabrication, and Test of a Vertical Attitude Takeoff and Landing Unmanned Air Vehicle*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1993.

[Twi94]   Twite, E., *Selection and Integration of a Global Positioning System with a CPU*, Masters Thesis, Naval Postgraduate School, Monterey, CA, June 1994.

[War85]   Ward, Paul T. and Mellor, Stephen J., *Structured Development for Real-Time Systems, Volumes I, II and III*, Yourdon Press, Englewood Cliffs, NJ, 1985.

[You89]   Yourdon, Edward, *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, NJ, 1989.

APPENDIX A: REAL TIME EXECUTIVE SOURCE CODE

```
/*************************** DEFS.H *********************************
      PROGRAM INITIALIZATION
***********************************************************************/
/* Note: MOXA-CL.obj and 812CL.lib must be linked into executable file */
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>                               /* For clrscr and cprint */
# include <alloc.h>                              /* For coreleft and malloc */
# include <dos.h>                            /* For DOS and BIOS interrupts */
# include <setjmp.h>                             /* For ctrl-break handler */

#include "c:\pcls-802\lib\c\head-c.h"          /* For PCL-744 definitions */

/***********************************************************************
      VARIABLE DEFINITIONS
***********************************************************************/

# define TRUE       1
# define FALSE      0

/* Used for RTC Timer */
# define RTC_INT      0x70                       /* RTC fires interrupt 70h */
# define RTC_INDEX    0x70              /* RTC Index Register I/O Address */
# define RTC_DATA     0x71               /* RTC Data Register I/O Address */
# define REG_A        0x0A
# define REG_B        0x0B
# define REG_C        0x0C
# define REG_D        0x0D
# define INT_FLAG     0x40              /* Periodic Interrupt Flag is bit 3 */
# define NMI_FLAG     0x80          /* Non-maskable Interrupt Flag bit 4 */
# define RATE_SET     0x0B          /* Used to set control cycles per sec: */
# define RATE         32                  /* 32768 << (RATE_SET - 1) */
# define PIC_STATUS   0xA1

/* Definitions for PCL-744 Serial I/O */
#define  IMUPORT        3                       /* Port number from IMU */
#define  SGPS_PORT      4               /* Port number for Slave GPS Rcvr */
#define  DLPORT         5                       /* Port number to Data Link */
#define  MGPS_PORT     10             /* Port number for Master GPS Rcvr */
#define  CR           0x0D             /* Carriage Return is ASCII 13h */
#define  LF           0x0A                    /* Line Feed is ASCII 10h */
#define  IOMODE (BIT_8 | P_NONE | STOP_1)          /* 8-N-1 (p. 12) */
#define  MODMODE   0x00                       /* DTR and RTS off (p.26) */
#define  HWMODE    0x00                   /* HW and SW flow ctrl off (p.33) */
```

/* Definitions for GPS routines are in GPSDEFIN.H. Each GPS module
   contains its own prototypes, included in the file below: */

```
#include "c:\borlandc\twitefin\gpsfun.h"
```

```
/**********************************************************************
     SUBROUTINE PROTOTYPES FOR MAIN PROGRAM (Table of Contents)
 **********************************************************************/
```

```
/* Variables for Serial I/O */
    struct T_GPS *gps;
    PANDL *imu_buf, *gps_buf, *gps_print, *dl_buf;


/* Variables for AtoD */
extern pcl812(int, unsigned int *);
unsigned int param[60];                  /* PCL-812 parameter array */
unsigned int data[20];                   /* Conversion data buffer  */
unsigned int far *dat;


/* Variables for Counter/Timer */
int datreg = 0x210;                      /*  Ctr/timer board, base address */
int conreg = 0x211;                      /*  Ctr/timer board, base addr +1 */
```

```
/****************************************************************
                Archytas Real-Time Executive Program            (Page 1)

        Author:     LT Peter M. Hoffman
        Written:    1 October 1993
        Revised:    1 June 1994
        Compiler:   Borland C++ 2.0

This RTE program provides the basis of the controller for the Archytas
Unmanned Air Vehicle.  Modifications to the flight_control() and
cmd_to_servos() procedures could adapt this controller to any UAV
using the same data path.

This controller is the center of a multi-dimensional inter-
disciplinary project collaborated by a number of students from
various departments of the Naval Postgraduate School, Monterey, CA.

Please see Thesis Document for complete details and explanation.
****************************************************************/

# include "c:\control\defs.h"                /* All definitions and prototypes */

int cyclecount = 0, vane_step = 0;
int thr_cmd, ail_cmd, elev_cmd, rud_cmd;
void interrupt new_vector(void);
void interrupt (*old_vector)();
int fmu_start_flag = FALSE;
jmp_buf cbreak_rtn;
```

```
void main(void)
{/***********************************************************************
    The main program initializes the system, then calls menu() to
    interface with the user for further actions.
 ***********************************************************************/

    /* Set up special exit handling routines */
    atexit(shut_down);
    ctrlbrk(break_handler);

    /* Set up structures to hold data */
    gps = malloc( sizeof( struct T_GPS));
    imu_buf = malloc( sizeof( PANDL ));
    imu_buf->ptr = calloc( 100, sizeof( char ));
    gps_buf = malloc( sizeof( PANDL ));
    gps_buf->ptr = calloc( 500, sizeof( char ));
    dl_buf = malloc( sizeof( PANDL ));
    dl_buf->ptr = calloc( 100, sizeof( char ));

    clrscr();

    /* Set up control-break resume point */
    if(setjmp(cbreak_rtn) != 0) {clrscr(); printf("\nRestarting Program...");}

    /* Begin user interface */
    printf("\t\tArchytas Monitor Program");
    printf("\n\nInitializing Hardware...");
    initialize_hw();
    menu();

} /* End Main */
```

```c
void menu(void)
{/********************************************************************
     This procedure interfaces with the user, querying for the desired
     response and invoking the appropriate routine.
 ********************************************************************/
     char ch;

     while(1) {
      printf("\n\nCommand ('?' for help): ");
      scanf("%s", &ch);
      switch (ch) {
      case 'c':                                          /* Check Hardware */
          check_hardware();
          break;
      case 's':                                              /* Start FMU */
          start_fmu();
          break;
      case 'q':                                               /* Quit FMU */
          quit_fmu();
          break;
      case 'f':                                             /* Flight Data */
          show_flight_data();
          break;
      case 'm':                                            /* Memory Dump */
          mem_dump();
          break;
      case 'r':                                       /* Display Registers */
          show_regs();
          break;
      case 'i':                                        /* Interrupt Vector */
          int_vector();
          break;
      case 'd':                                           /* DOS command */
          dos_cmd();
          break;
      case '?':                                        /* List Alternatives */
          printf("\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
          "The following are valid commands:",
          "--------------------------------",
          "(c)heck hardware",
          "(s)tart flight management unit",
          "(q)uit flight management",
          "(f)light data menu",
          "(m)emory contents display",
          "(r)egister contents display",
          "(i)nterrupt vector display",
          "(d)os command",
          "(t)erminate program");
          break;
```

```
        case 't':                                      /* QUIT */
            printf("\nTerminating Program...");
            exit(0);
        default:
            puts("\nNot a valid command.  Type '?' for help.");
            } /* End Switch */
        } /* End While */
    }   /* End Menu */
```

```
void initialize_hw( void )
{/*******************************************************************
    This procedure initializes the PCL-812, PCL-744 and PCL-830
    hardware boards for UAV controller operation.
 ********************************************************************/

    int   rtn_code, port, i;
    int   gpstart[]= {'@','@','B','a',0x01,0x32,0x0D,0x0A};

    /* PCL-812 A/D Board Initialization ********************************/
    /* Note: Jumpers on the PCL-812 must be set as follows:
         I/O Port Address (SW1): 220h, 0 Wait States
         Trigger Mode    (JP1): Internal
         IRQ Level       (JP4): 5
         A/D Input Range (JP9): +/- 5V
         Parameter Array as follows: */

    dat = data;
    param[0]  = 0;                          /* Board number */
    param[1]  = 0x220;                      /* Base I/O address */
    param[4]  = 5;                          /* IRQ level   : IRQ5 */
    param[5]  = 50;           /* Pacer rate = 2M / (50 * 100) = 400 Hz */
    param[6]  = 100;
    param[7]  = 0;                   /* Trigger mode: internal pacer trigger */
    param[8]  = 0;                          /* Non-cyclic mode */
    param[10] = FP_OFF(dat);         /* Offset of A/D data buffer A    */
    param[11] = FP_SEG(dat);         /* Segment of A/D data buffer A */
    param[12] = 0;            /* Data buffer B offset: 0 if not used    */
    param[13] = 0;            /* Data buffer B segment: 0 if not used */
    param[14] = 5;                   /* A/D conversion number        */
    param[15] = 0;                   /* A/D conversion start channel */
    param[16] = 5;                   /* A/D conversion stop channel  */
    param[17] = 0;                   /* Overall gain code, 0 : +/- 5V */

    /* param[18] = FP_OFF(gain_array);    FYI: Output Registers
         param[19] = FP_SEG(gain_array);
         param[45] : Error code
         param[46] : Return value 0
         param[47] : Return value 1 */

    pcl812(3, param);                  /* Func 3 : Hardware initialization */
    if (param[45] != 0) {
            printf("\n PCL-812 Driver Initialization Failed!");
            exit(1);
            }

    pcl812(4, param);                     /* Func 4 : A/D initialization */
    if (param[45] != 0) {
            printf("\nA/D Initialization Failed!");
            exit(1);
            }
```

```
/********************** PCL-744 Initialization ***********************/
for (port = 3; port <= 10; port++) {                        /* Set up each port */
    printf("\nConfiguring port number %d (Cable #%d):", port, port-2);

    /* Set I/O control params */
    rtn_code = sio_ioctl( port, B9600, IOMODE );
        if (rtn_code != 0)
            printf("\nI/O control error on port %d.", port);

    /* Set line control parameters */
    rtn_code = sio_lctrl( port, MODMODE );
        if (rtn_code != 0)
            printf("\nLine control error on port %d", port);

    /* Set flow control params */
    rtn_code = sio_flowctrl( port, HWMODE );
        if (rtn_code != 0)
            printf("\nFlow control error on port %d", port);

    /* Last, open the port which enables it for I/O */
    rtn_code = sio_open( port );
        if (rtn_code != 0)
            printf("\nError opening port %d", port);

} /* End For port++ Loop */

/* Send 'I': tell IMU to begin sending */
rtn_code=sio_putch( IMUPORT, 'I');
if ( rtn_code == 1) printf("\n   IMU initialized OK");
else printf("\n   IMU NOT initialized");

/* Initialize GPS to send position msg every sec */
rtn_code=sio_putb( SGPS_PORT, gpstart, 8);
if ( rtn_code <= 0 ) printf("\n   GPS NOT initialized");
if ( rtn_code == 8 ) printf("\n   GPS initialized OK");

/* Set Tx/Rx timeout to 1 second */
rtn_code = sio_timeout( 18 );

/* Flush Rx and Tx Buffers */
sio_flush( SGPS_PORT, 2 );
sio_flush( IMUPORT, 2 );
sio_flush( DLPORT, 2 );
```

71

```
/*** PCL-830 Initialization (Am9513A chip) *****************************/
/*      This portion of initialize() written by Pat Moran.          */
/*      All values are decimal, but represent binary register settings. */
/*      See Am9513A Technical Manual for details                    */

outportb(conreg,255);                          /* Reset all board functions */
outportb(conreg,23);                           /* Select master mode register */
outportb(datreg,176);               /* Low byte enables FOUT, F1 source */
outportb(datreg,65);                /* Hi byte selects binary division    */
                                    /* Disable increment, 8 bit bus */
                                    /* FOUT on, divide by 1. */
                                    /* RTE, Page 7 */


outportb(conreg,249);               /* Diable prefetch for write ops */
for (i=1;i<=5;i++)
 {
     outportb(conreg,i);                        /* Select ctrs 1-5 */
     outportb(datreg,98);           /* Low byte: set modes of ctrs 1-5 in CMR */
     outportb(datreg,27);           /* High byte: no gating for ctrs 1-5      */
 }
for (i=25;i<=29;i++)
 {
     outportb(conreg,i);            /* Load hold registers for refresh rate */
     outportb(datreg,0);            /* Load + Hold = Refresh Rate */
     outportb(datreg,31);           /* This combo gives 25 ms rate (40 Hz) */
 }
for (i=9;i<=13;i++)
 {
     outportb(conreg,i);            /* Select load registers for pulse width */
     outportb(datreg,103);          /* Sets time for next pulse */
     outportb(datreg,5);
 }
for (i=233;i<=237;i++) outportb(conreg,i);
     outportb(conreg,i);            /* Load & arm ctrs 1-5 */

outport(conreg,127);
printf("\nCompleted Initialization of Servos.");

} /* End Initialize_HW */
```

```
void check_hardware(void)
{/*********************************************************************
     This procedure checks that all the hardware is properly
     configured and operational.
 **********************************************************************/

     int   i, card_type = 0x744, card_no = 1, rtn_code, port;
     char  *buf = "Test String";
     union REGS xreg, yreg;
     unsigned elist, drives=0, ports=0, printers=0;

     elist = biosequip();                          /* Determine BIOS Equipment */
     if (elist & 0x0001) drives = ((elist & 0x00c0) >> 6)+1;
     ports = (elist & 0x0e00) >> 9;
     printers = (elist & 0xc000) >> 14;
     printf("\nThis system has %d diskette drives, %d serial ports, %d printer ports ",\
              drives, ports, printers);
     if ((elist & 0x0002) >> 1) printf("and a math co-processor.");

     printf("\nIt has %uK of RAM and %lu stack available.", \
                  biosmemory(), coreleft());

     rtn_code = sio_bank(card_type, card_no);   /* Display address of PCL-744 card */
     printf("\nThe PCL-744 card is mapped to address %Fp", rtn_code);

     rtn_code = sio_id( card_type, card_no);      /* Display ID number of 744 card */
     printf("\nIt is card number %d", rtn_code);

     for (port = 3; port <= 10; port++) {                        /* Set up each port */
         printf("\n\nChecking port number %d (Cable #%d):", port, port-2);

         /* Check I/O Control Params */
         rtn_code = sio_getbaud( port );
         printf("\nBaud rate of port %d set to %d", port, rtn_code);

         rtn_code = sio_getmode( port );
         printf("\nMode of port %d set to %d.", port, rtn_code);

         /* Check Line Control Params */
         rtn_code = sio_lstatus( port);
             if (rtn_code < 0)
                 printf("\nError in status for port %d", port);
             else
                 printf("\nModem line status of port %d is %d.", port, rtn_code);

         /* Check Flow Control Params */
         rtn_code = sio_getflow( port );
         printf("\nHardware flow control of port %d set to %d.", port, rtn_code);
```

```
       /* Do loopback test */
      if (sio_loopback( port, buf, 12 ) == 0)
          printf("\nPort %d loopback test OK.", port);
      else
          printf("\nPort %d failed loopback test.", port);

      printf("\n\nReview data for port above.\nPress any key to continue.");
      getch(); /* Wait for key */
    } /* End For port++ Loop */

    printf("\n\nParameter Array for PCL-830 board set to:");
    for (i = 0; i <= 17; i++) {
        if (i==10 || i==11) printf("\nparam[%3d] = %Fp", i, param[i]);
        else printf("\nparam[%3d] = %d", i, param[i]);
    }
} /* End Check_Hardware */
```

```c
void start_fmu(void)
(/***********************************************************************
    This procedure sets up the real-time clock to provide periodic
    interrupts at 64 Hz which will trigger the flight management unit.
************************************************************************/
    unsigned char value, bit_set, new_value;

    if (fmu_start_flag == TRUE) {                    /* Check if already started */
        printf("\nThe FMU has already been started.");
        return;
    }

    printf("\n\n Starting the Flight Management Unit.");

    /* Get old vector number for posterity */
    old_vector = getvect(RTC_INT);
    printf("\nThe address of the old vector is: %Fp\n", old_vector);

    /* Now set RTC to generate interrrupt at rate set in DEFS.h */
    /* Alter interrupt rate to new rate (32768 >> RATE_SET - 1) */
    value = ReadRTC(REG_A);                          /* Read register A */
    bit_set = value & 0xF0 | RATE_SET;        /* Lowest 4 bits sets rate of int */
    SetRTC(REG_A, bit_set);                   /* Set to new rate of periodic int */

    new_value = ReadRTC(REG_A);
    printf("\nReg A was %x, now %x with new rate set.",value, new_value);

    /* Enable periodic interrupts with the RTC */
    disable();
    value = ReadRTC(REG_B);                          /* Read register B */
    bit_set = value | INT_FLAG;               /* Enable periodic interrupts */
    SetRTC(REG_B, bit_set);                   /* on IRQ 8 (Int 70).        */

    new_value = ReadRTC(REG_B);
    printf("\nReg B was %x, now %x with int flag set.",value, new_value);

    /* Change interrupt vector to run my program */
    disable();                                /* Disable interrupts when changing */
    setvect(RTC_INT, new_vector);
    enable();
    printf("\nInstalled new vector: %p\n", new_vector);

    value = ReadRTC(REG_C);              /* Clear pending int by reading reg */
```

```
    /* Initialize PIC to enable interrupts */
    value = inportb(PIC_STATUS);                    /* Read PIC Status Register */
    bit_set = value & 0xfe;
    outportb(PIC_STATUS, bit_set);                  /* Clear bit 0 to enable ints */

    new_value = inportb(PIC_STATUS);
    printf("\nPIC mask was %x, now %x with bit 0 cleared.",value, new_value);
    enable();

    fmu_start_flag = TRUE;
}  /* End Start_FMU */
```

```
void quit_fmu(void)
{/*********************************************************************
    This procedure stops the periodic interrupt, effectively halting the
    flight management unit, and resets the real-time clock chip back to
    its original configuration.
**********************************************************************/
    unsigned char value, bit_set, new_value;

    if (fmu_start_flag == FALSE) {              /* Make sure it has been started */
        printf("\nThe fmu has not yet been started.");
        return;
    }
    else {
    printf("\n\n Stopping the Flight Management Unit.");

    /* Put system back to normal */
    /* First clean up RTC */
    disable();                                  /* Disable interrupts while changing */

    /* Clear periodic interrupt bit */
    value = ReadRTC(REG_B);
    bit_set = value & 0xBF;
    SetRTC(REG_B, bit_set);

    new_value = ReadRTC(REG_B);
    printf("\nReg B was %x, now %x with int flag clrd.",value, new_value);

    /* Reset rate to 1024 Hz */
    value = ReadRTC(REG_A);
    bit_set = value & 0xF0 | 0x06;
    SetRTC(REG_A, bit_set);

    new_value = ReadRTC(REG_A);
    printf("\nReg A was %x, now %x with new rate set.",value, new_value);

    /* Reset interrupt vector to original value */
    setvect(RTC_INT, old_vector);
    enable();
    printf("\nThe cyclecount is: %d\n", cyclecount);

    fmu_start_flag = FALSE;
    } /* End Else */
} /* End Quit_FMU */
```

```
unsigned char ReadRTC( unsigned char reg )
/****************************************************************************
    This function returns the value of the specified register on the
    real-time clock chip.
 *****************************************************************************/
    unsigned char reg_nmi, value;

    reg_nmi = reg | NMI_FLAG;                /* Disable Non-Maskable Int */
    outportb (RTC_INDEX, reg_nmi);           /* Tell CMOS which reg to read */
    value = inportb (RTC_DATA);                   /* Read value of register */
    return value;
}  /* End Read_RTC */


void SetRTC( unsigned char reg, unsigned char value )
/****************************************************************************
    This procedure sets a new value into the specified register
    of the real-time clock chip.
 *****************************************************************************/
    unsigned char reg_nmi;

    reg_nmi = reg | NMI_FLAG;                /* Disable Non-maskable Int */
    outportb (RTC_INDEX, reg_nmi);           /* Tell CMOS which reg to set */
    outportb (RTC_DATA, value);                   /* Write value to register */
}  /* End Set RTC */


void interrupt new_vector()
/****************************************************************************
    This is the flight management unit procedure that is run on each
    occurrence of the periodic interrupt.
 *****************************************************************************/
    cyclecount++;                                   /* Count number of cycles */
    cyclecount %= RATE*10;               /* Normalize count every 10 seconds */
    reset_int();                         /* Reset Interrupt to enable next one */
    execute_cycle();                          /* Do something constructive */
}  /* End New_Vector */


void reset_int(void)
/****************************************************************************
    This procedure resets the real-time clock chip and the PIC chips
    in order to facilitate another periodic interrupt.
 *****************************************************************************/
    unsigned char value;

    value = ReadRTC(REG_C);            /* Must read reg C to get another int */

    disable();
    outportb(0x0a0, 0x20);             /* Send non-specific EOI to slave PIC */
    outportb(0x20, 0x20);              /* and master PIC                     */
    enable();
}  /* End Reset_Int */
```

```c
void execute_cycle(void)
{/********************************************************************
    This procedure is the heart of the controller.  It is the routine
    that is invoked during every occurrence of the real-time clock
    interrupt, coordinating the execution of other modules which
    comprise the control and communication processes of the UAV.
    ********************************************************************/
    unsigned char value, bit_set;
    int imu_ok, gps_ok, dl_ok;

    value = inportb (0x61);
    bit_set = value ^ 0x02;                 /* Toggle speaker enable bit */
    outportb(0x61, bit_set);            /* (Sounds like the motor is running) */

/* sio_putb( DLPORT, "GPS: ", 5);
    Slave_gps( gps );                         Calls to Eric Twite's Stuff
    xmit_to_gnd( &gps->out );                   (Not yet operational)
    free( gps->out.ptr );                              *****/

    dl_ok = read_datalink( dl_buf );        /* Read uplink every cycle  */
    /* Put code to deal with info from datalink uplink here           */

    if (cyclecount % 4 == 0) {              /* Read IMU every 4th cycle */
        imu_ok = read_imu( imu_buf );           /* Send every 0.5 sec  */
        if ( cyclecount % (RATE/2) == 0 && imu_ok ) {
            sio_putb( DLPORT, "IMU: ", 5);      /* IMU label in data stream */
            xmit_to_gnd( imu_buf );
        }
    }
    if(cyclecount % (42) == 0) {               /* Read GPS every 1.3 sec */
        gps_ok = read_gps( gps_buf );
        if (gps_ok) {                         /* If full msg rcvd,        */
            sio_putb( DLPORT, "GPS: ", 5);    /* also send to ground    */
            xmit_to_gnd( gps_buf );           /* with data stream label */
        }
    }
    read_atod();                           /* Read AtoD every cycle */

    /* Last, with all flight data in hand, control aircraft */
    flight_control( &thr_cmd, &ail_cmd, &elev_cmd, &rud_cmd);
    cmd_to_servos( thr_cmd, ail_cmd, elev_cmd, rud_cmd);

} /* End Execute_Cycle */
```

```
int read_imu( PANDL *buffer )
{/*******************************************************************
    This procedure reads into a pre-established buffer the data
    from the onboard Inertial Measurement Unit.
 ********************************************************************/
    int eol = CR;
    int queue;

    queue = sio_iqueue( IMUPORT );
    if (queue > 100) queue = 100;          /* Truncate to size allocated in main */
    if (queue > 38) {                      /* Buffer has at least 1 full + partial msg */
        buffer->len = sio_linput( IMUPORT, buffer->ptr, queue, eol );
        buffer->len = sio_read( IMUPORT, buffer->ptr, 38);
    }
    else if (queue > 0)                    /* or has at most 1 full msg (usual condition) */
        buffer->len = sio_linput( IMUPORT, buffer->ptr, queue, eol);
    else
        buffer->len = 0;                             /* or has nothing in the buffer */
    if (buffer->len == 38) return TRUE;        /* Test if message is complete*/
    else return FALSE;
} /* End Read_IMU */


int read_gps( PANDL *buffer )
{/*******************************************************************
    This procedure reads into a pre-established buffer the data
    from the onboard Global Positioning System.
 ********************************************************************/
    int eol = LF, queue;

    queue = sio_iqueue( SGPS_PORT );                 /* How long is rcv queue? */
    while (queue > 135) {                    /* Pare down to last 2*68-1 chars */
        queue -= 135;                        /*    (At most 1 full message)    */
        if (queue < 68) queue += 68;         /* (But at least 1 full message)  */
        if (queue > 500)                     /* Max buffer space 500 bytes     */
            buffer->len = sio_read( SGPS_PORT, buffer->ptr, 500);
        else
            buffer->len = sio_read( SGPS_PORT, buffer->ptr, queue);
        queue = sio_iqueue( SGPS_PORT );
    }
    /* Now, at most 1 full msg exists in the queue + maybe a partial msg */
    if (queue > 68)                          /* If partial msg exists, read it away  */
        buffer->len = sio_linput( SGPS_PORT, buffer->ptr, queue, eol );

    /* Now, only 1 full msg should exist in the queue, so read it */
    buffer->len = sio_linput( SGPS_PORT, buffer->ptr, queue, eol );

    if (buffer->len == 68) return TRUE;          /* Test to make sure full msg */
    else return FALSE;
} /* End Read_GPS */
```

```c
void read_atod(void)
/***************************************************************
    This procedure calls PCL-812 intrinsic function 5, which triggers an
    A/D conversion on analog data inputs as set in the param array.
 ***************************************************************/
    /* Record A/D Conversions */
    pcl812(5, param);                    /* Func 5 : Pacer trigger A/D conversion */
    if (param[45] != 0)              /*              with software data transfer  */
        printf("\nA/D Conversion Failed!");
}   /* End Read_AtoD */



void xmit_to_gnd( PANDL *buffer)
/***************************************************************
    This procedure transmits the contents of the buffer to the ground
    through the datalink.
 ***************************************************************/
    int strglen, txbuff;

    if (buffer->len > 0) {
        txbuff = sio_ofree( DLPORT );          /* Get free space in xmit buffer */
        if (buffer->len < txbuff) {            /* If enough buffer space, send   */
            strglen = sio_putb(DLPORT, buffer->ptr, buffer->len);
            if (strglen == 0) {                        /* Else                    */
                sio_flush(DLPORT, 1);              /* Get rid of the old data      */
                strglen = sio_write(DLPORT, "WARNING: Buffer cleared! ", 25);
                strglen = sio_write(DLPORT, buffer->ptr, buffer->len);
            }
        }
    }
}   /* End Xmit_to_Gnd */



int read_datalink( PANDL *buffer )
/***************************************************************
    This procedure reads the contents of the datalink's receive buffer
    containing information sent from the ground through the datalink.
 ***************************************************************/
    int queue, eol = '#';

    queue = sio_linput(DLPORT, buffer->ptr, 100, eol);    /* Get queue length */
    if (queue > 0) {                            /* If something is in queue, read it */
        queue = sio_read(DLPORT, buffer->ptr, 2);      /* Read length of msg */
        buffer->len = atoi(buffer->ptr);           /* Convert length to an integer */

        /* Read in buffer of specified length */
        queue = sio_read(DLPORT, buffer->ptr, buffer->len);
        return TRUE;
    }
    else return FALSE;                    /* If nothing waiting in buffer, continue */
}   /* End Read_Datalink */
```

```
void flight_control(int *thr, int *ail, int *elev, int *rud)
{/**********************************************************************
      This procedure is a place holder for the actual control algorithm
      being designed by the Aeronautical Engineering Department.

      The procedure envisioned here will perform appropriate data filtering
      and will use the filtered data to calculate the necessary control
      surface positions.  The output of this procedure is the angle of each
      of the standard control surfaces.  The cmd_to_servos procedure will
      convert these standard control surfaces into individual control vane
      angles.  This conversion will differ depending on the mode of flight,
      whether vertical or horizontal.
      **********************************************************************/
      /* Get or calculate pilot commands  */
      /* Calculate control surface inputs  */

      /* The steps below are just a demo to exercise the servos to their
            full extension in increments of 2 degrees until replaced by the
            actual control algorithm  */

      *thr = 100;                              /* Throttle stays constant */
      vane_step += 2;                          /* Increase vanes 2 deg each cycle  */
      vane_step %= 200;                        /* All vanes go -30 to +30 deg */
      *ail = vane_step;
      *elev = vane_step;
      *rud = vane_step;

      /* Delete global variable step_vane when this test routine deleted */
}  /* End Flight_Control */
```

```c
void cmd_to_servos(thr, ail, elev, rud)
{/***********************************************************************/
/*  Written by LCDR Pat Moran              5/14/93                     */
/*  Originally called chgangle() -- see thesis description             */
/*  Basic PWM routine by LT Paul Merz [Mer92]                          */
/*  Demo to move aileron, rudder, elevator, & throttle from 2 joysticks. */
/*  Blends 3 degrees-of-freedom into 4 independent vane commands.      */
/***********************************************************************/

    int i,hibyte,lobyte,angle,vane[5];

    vane[0] = ail/4 + rud/2;              /* V1;  Translation algorithm fm 3 */
    vane[1] = ail/4 + elev/2;             /* V2;  control surfaces to 4 vanes */
    vane[2] = ail/4 - rud/2;              /* V3;                             */
    vane[3] = ail/4 - elev/2;             /* V4;                             */
    vane[4] = thr;                        /* Throttle needs no conversion */
    outportb(conreg,223);                 /* Disarm counters 1-5 */

    for (i=0;i<=4;i++) {
        angle=((1900/206)*(vane[i])+600); /* Convert fm deg to dig # */
        hibyte=(angle/256);               /* Calc high byte, residue left */
        lobyte=(angle-hibyte*256);        /* Calc low byte fm residue */
        outportb(conreg,(i+9));           /* Load counters 1-5 */
        outportb(datreg,lobyte);          /* Load low byte */
        outportb(datreg,hibyte);          /* Load high byte */
    }

    for (i=233;i<=237;i++)
        outportb(conreg,i);               /* Set toggle high for counters 1-5 */

    outportb(conreg,127);                 /* Load & arm counters 1-5 */
} /* End Cmd_to_Servos */
```

```c
void show_flight_data(void)
/*********************************************************************
    This procedure queries the user to determine which flight data to
    display and calls the appropriate routine.
*********************************************************************/
    char ch;

        printf("\n%s\n%s\n%s\n%s\n%s\n%s\n\n%s",
        "Display which data?",
        "-------------------",
        "(g)ps position",                       /* Can list other gps data here too */
        "(i)mu data",
        "(a)ir data",
        "(s)ervo positions",
        "Choice: ");
    scanf("%s", &ch);
    switch (ch) {
    case 'g':                                   /* GPS Position */
        show_gps_posit();
        break;
    case 'i':                                   /* IMU Data */
        show_imu();
        break;
    case 'a':                                   /* Analog Air Data */
        show_air_data();
        break;
    case 's':                                   /* Servo Position */
        show_servo_posit();
        break;
    default:
        printf("\nData choice not recognized!");
        return;
    } /* End Switch */
}   /* End Show_Flight_Data */


void show_imu(void)
/*********************************************************************
    This procedure prints the most recently acquired IMU data.
*********************************************************************/
    int i;

    printf("\nLatest IMU data: %d characters.\n", imu_buf->len);
    for (i = 0; i < imu_buf->len; i++) {
        putchar(imu_buf->ptr[i]);
        if (i % 4 == 0) putchar(' ');
    }
}  /* End Show_IMU */
```

```c
void show_gps_posit(void)
{/*************************************************************************
    This procedure prints the most recently acquired GPS data.
 *************************************************************************/
    int i;

    /* To print from Twite's GPS structure, when complete */
/*  printf("\nLatest GPS position:\nLat: %d.%d N, Long: %d.%d W",\
    gps->pcs.latitude.degrees, gps->pcs.latitude.minutes, \
    gps->pcs.longitude.degrees, gps->pcs.longitude.minutes);
*/
    /* To print from gps_buf raw data buffer */
    gps_print = Bin_to_ascii( gps_buf, 4);          /* Convert to ASCII chars */
    printf("\nLatest GPS data: %d / %d characters.\n", \
                    gps_buf->len, gps_print->len);
    for (i = 0; i < gps_print->len; i++) putchar(gps_print->ptr[i]);
    free(gps_print->ptr);
    free(gps_print);
} /* End Show_GPS_Posit */


void show_air_data(void)
{/*************************************************************************
    This procedure prints the most recently acquired A/D data.
 *************************************************************************/
    unsigned int i;
    float DataBuf;

    /* Calculate analog values from raw data in data array */
    for (i = 0; i < param[16]; i++) {
        DataBuf = data[i] & 0xFFF;
        DataBuf = (10 * DataBuf / 4096) + (-5);
            /* Calculations:
                    10          : A/D input range (-5V to 5V)
                    4096        : Full scale 12 bit A/D data
                    DataBuf     : A/D input data masked to 12 bits
                    (-5)        : A/D input base "-5" V
            */
        printf("\ndata[%3d] = % 1.2f V returned as %x", i, DataBuf, data[i]);
    } /* End For all data entries */
} /* End Show_Air_Data */


void show_servo_posit(void)
{/*************************************************************************
    This procedure prints the present position of all servos.
 *************************************************************************/
    printf("\nThr: %d, Ail: %d, Elev: %d, Rud: %d.", \
                thr_cmd, ail_cmd, elev_cmd, rud_cmd);
} /* End Show_Servo_Posit */
```

```c
void close_ports (void)
/******************************************************************
    This procedure closes all serial ports on the PCL-744 card and
    flushes transmit and receive buffers for each port.
******************************************************************/
    int port, rtn_code;

    for (port = 3; port <= 10; port++) {
        rtn_code = sio_close( port );
        if (rtn_code != 0)
            printf("\nError closing port %d", port);
        else
            printf("\nClosed port %d", port);
        sio_flush( port, 2);
    } /* End For all ports */
} /* End Close_Ports */


void shut_down(void)
/******************************************************************
    This procedure is invoked at program exit to ensure that the system,
    including communication ports, ISR vectors, and allocated memory, is
    properly terminated and returned to its normal operating configuration.
******************************************************************/
    quit_fmu();              /* Stop the flight management unit */
    close_ports();           /* Close and flush all ports */
    free(gps);               /* Free all globally allocated memory */
    free(imu_buf->ptr);
    free(imu_buf);
    free(gps_buf->ptr);
    free(gps_buf);
    free(dl_buf->ptr);
    free(dl_buf);
} /* End Shut_Down */
```

```
void int_vector(void)
{/*********************************************************************
    This procedure prints the address registered for the ISR of the
    interrupt number given by the user.
 *********************************************************************/
    void interrupt (*int_handler)();
    int intno;

    printf("\nEnter interrupt number in hex: ");
    scanf("%x", &intno);
    int_handler = getvect(intno);
    printf("\nThe address of the handler is: %Fp\n", int_handler);
}   /* End Int_Vector */



void mem_dump(void)
{/*********************************************************************
    This procedure prints the values of a given portion of memory.
 *********************************************************************/
    int i, n;
    char far *far_ptr;

    printf("\n\nEnter begin memory address to dump (eg. F000:E000): ");
    scanf(" %p", &far_ptr);
    printf("\nHow many bytes to display? ");
    scanf("%d", &n);

    printf("\nDump of %d bytes at %Fp\r\n", n, far_ptr);
    for(i=0; i<n; i++) {
        printf("\n%Fp   %Fx", (far_ptr+i), *(far_ptr+i));
    }
} /* End Mem_Dump */



void show_regs(void)
{/*********************************************************************
    This procedure prints the current values of all CPU and segment
    registers.
 *********************************************************************/
    union REGS xr, yr;
    struct SREGS sr;

    segread(&sr);
    printf("\nax = %x, bx = %x, cx = %x, dx = %x", \
                xr.x.ax, xr.x.bx, xr.x.cx, xr.x.dx);
    printf("\nsi = %x, di = %x, cflag = %x, flags = ", \
                xr.x.si, xr.x.di, xr.x.cflag);
    bit_print(xr.x.flags);
    printf("\ncs = %x, ds = %x, es = %x, ss = %x", \
                sr.cs, sr.ds, sr.es, sr.ss);
} /* End Show_Regs */
```

```
void bit_print(unsigned int v)
{/********************************************************************
    This procedure prints the binary representation of the given
    hexidecimal number.
 *********************************************************************/
    int i, mask = 1 << 15;

    printf("%x = ", v);
    for (i = 1; i <= 16; i++) {
        putchar(((v & mask) == 0) ? '0' : '1');
        v <<= 1;
        if (i % 4 == 0) putchar(' ');
    }
}  /* End Bit_Print */


void dos_cmd(void)
{/********************************************************************
    This procedure invokes a DOS shell.
 *********************************************************************/
    char cmd[40];

    printf("\nDOS COMMAND:> ");
    gets(cmd);
    system(cmd);

}  /* End DOS_Cmd */
```

```
int break_handler(void)
{/***********************************************************************

    This procedure is invoked upon a control-break or control-c sequence
    from the keyboard.  It gives the user more flexibility in determining
    how extensively he desires to reset the system.
 ***********************************************************************/

    char ch;
    union REGS xreg, yreg;

        printf("\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s",
        "Why did you break?",
        "--------------------------------",
        "(c)old reboot machine",
        "(w)arm reboot machine",
        "(r)estart program (reinitialize hardware)",
        "(g)o back to main menu",
        "(t)erminate program",
        "Choice: ");
    scanf("%s", &ch);
    switch (ch) {
    case 'c':                                         /* Cold Reboot */
        bootstrap(0);
        break;
    case 'w':                                         /* Warm Reboot */
        bootstrap(1);
        break;
    case 'r':                                      /* Restart Program */
        shut_down();
        longjmp(cbreak_rtn, 1);
        break;
    case 'g':                                           /* Main Menu */
        menu();
        break;
    case 't':                                               /* QUIT */
        printf("\nTerminating Program...");
        exit(0);
    default:
        menu();
    } /* End Switch */
}  /* End Break_Handler */
```

```c
void bootstrap(int input)
{
    union REGS reg;
    void (far *reboot)(void);
    int far *boottype;

    /* Set Far Pointers to the Boot Sector */
    FP_SEG(reboot) = 0xffff;
    FP_OFF(reboot) = 0;
    FP_SEG(boottype) = 0x40;
    FP_OFF(boottype) = 0x72;

    /* Issue a DOS disk reset request to flush caches*/
    reg.h.ah = 0x0d;
    int86(0x21, &reg, &reg);

    /* Set boot type and execute reboot */
    *boottype = (input ? 0x1234 : 0);    /* 0 = Cold, 1 = Warm */
    (*reboot)();
} /* End Bootstrap */

/* End of Real-Time Executive Program */
```
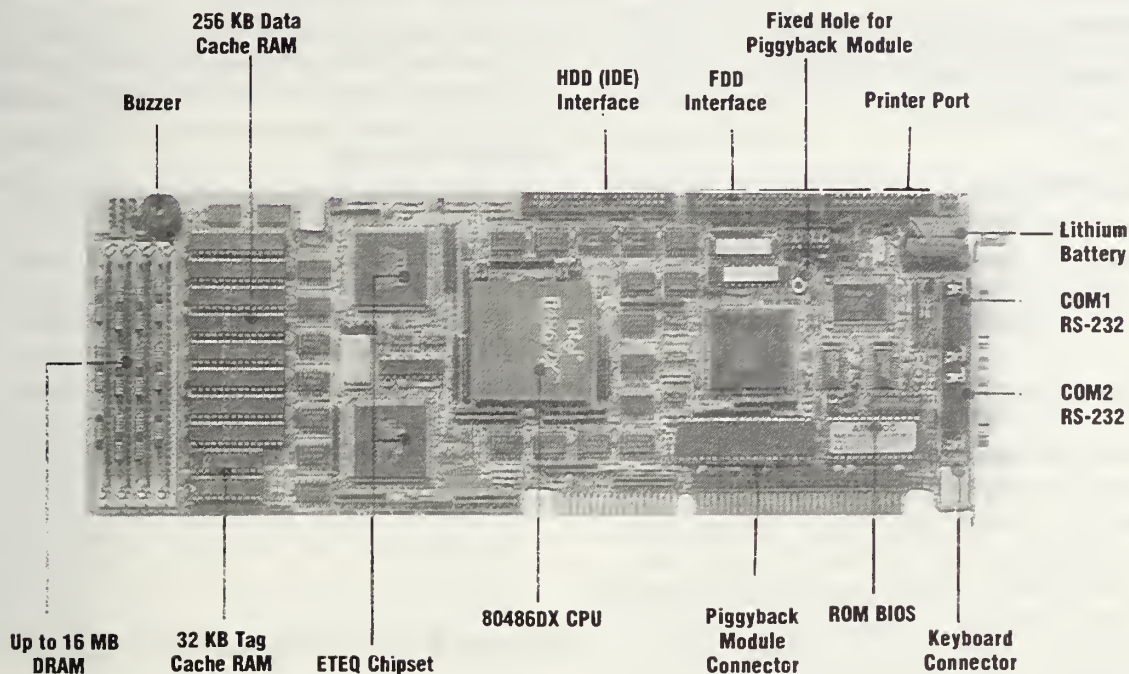
APPENDIX B: LIST OF VARIABLES

The following is a reference listing of the definitions of all variables used in the Real-Time Executive source code shown in Appendix A. (G) indicates a globally maintained variable.

angle	Value of digital representation of servo command angle
bit_set	Bit value to be written to register from the Real-Time Clock (RTC) chip
***boottype**	Pointer to memory location specifying hard or cold boot
buffer->len	Lenght field of PANDL structure. Represents the length of the buffer.
buffer->ptr	Pointer field of PANDL structure. Points to the actual data buffer.
cbreak_rtn	Pointer to position in program to return to after control-break (G)
ch	Variable to hold user response to menu selection
cyclecount	A counter incremented on every cycle of a control loop (G)
***dat**	The pointer to the data array (G)
data[20]	An array in which the PCL-812 stores the results of its A/D conversions (G)
DataBuf	Value of actual voltage calculated from A/D conversion data
dl_buf	PANDL to store raw data received through the datalink (G)
dl_ok	Boolean set true if message received on datalink
eol	Character which ends complete message from IMU or GPS
***far_ptr**	Pointer to memory location to begin inspection
fmu_start_flag	Boolean variable toggled on when FMU is started (G)
gps_buf	PANDL to receive raw data retrieved from GPS receiver (G)
gps_start	Array to hold start sequence to be sent to GPS receiver
gps_ok	Boolean set true if GPS message received is complete
hibyte	Value of high byte given to PCL-830 for the PWM signal
i	Loop increment used in various places
imu_buf	PANDL to receive raw data retrieved from IMU (G)
imu_ok	Boolean set true if IMU message received is complete
intno	Number of interrupt for which requesting ISR vector address
lobyte	Value of low byte given to PCL-830 for the PWM signal
***old_vector**	Pointer to hold value of old interrupt ISR vector (G)
new_value	Value of register read from RTC after a modification
param[60]	An array of parameters used to configure the PCL-812 board (G)
port	For loop increment to configure all ports
queue	Length of receive queue buffer from IMU or GPS
***reboot**	Pointer to memory location to jump to causing reboot
reg_nmi	Value of register read from the RTC with Non-Maskable Interrupt (NMI) bit set
rtn_code	Code returned from execution of PCL-744 I/O program
sr, xr, yr	Values read from computer internal registers
strglen	Number of characters transmitted to datalink buffer
txbuff	Length of free space available in datalink transmit buffer
value	Value of register read from the RTC
vane	Array holding vane servo commands for each of the servos
vane_step	Temporary variable holding command for control vane position (G)

APPENDIX C: HARDWARE DATA SHEETS

PCA-6146 All-in-One 486 CPU Card with Cache

Diagram labels:

- 256 KB Data Cache RAM
- Buzzer
- HDD (IDE) Interface
- FDD Interface
- Fixed Hole for Piggyback Module
- Printer Port
- Lithium Battery
- COM1 RS-232
- COM2 RS-232
- Up to 16 MB DRAM
- 32 KB Tag Cache RAM
- ETEQ Chipset
- 80486DX CPU
- Piggyback Module Connector
- ROM BIOS
- Keyboard Connector

Introduction

We designed the PCA-6146 for users who require high speed system performance in their industrial PC applications. The card is available in five 80486 CPU versions: 80486SX-25, 80486SX-33, 80486DX-33, 80486DX2-50 or 80486DX2-66. The card's all-in-one design includes memory caching, disk drive controllers, a watchdog timer and serial/parallel ports. We give every card a 24-hour dynamic burn-in test to ensure component reliability in harsh environments at temperatures up to 140°F (60°C).

With the PCA-6146 plugged into your passive backplane, your industrial PC becomes a true 32-bit 80486 compatible computer system.

The card's highly compact size, numerous features and unmatched cost/performance ratio make it ideal for high-end industrial applications where high CPU speed, minimum space and short MTTR are crucial.

Each PCA-6146 ships with either an 80486DX-33 MHz, 80486DX2-50 MHz or an 80486DX2-66 MHz CPU. These state-of-the-art CPUs feature an on-chip math coprocessor and an 8 KB cache memory for floating point calculations and fast memory access. 256 KB of 2nd-level cache memory allows the card to run at Landmark speeds in excess of 150 MHz.

Other standard features include two RS-232 serial ports, one parallel/printer port, an IDE hard disk drive interface, a floppy disk controller, a watchdog timer, piggyback module connectors and an on-board keyboard connector. You can configure system memory to anywhere from 1 MB to 16 MB using 256 KB, 1 MB or 4 MB SIMM DRAM in the PCA-6146's four memory sockets.

Features

- Completely 80486 PC/AT compatible
- 32 to 140°F (0 to 60°C) operating temperature
- Watchdog timer
- Optional Flash/RAM/ROM Disk Piggyback Module (PCD-8931) and/or Flat-panel/CRT VGA Piggyback Module (PCA-6443) install on the piggyback connector
- 80486 processor and AMI BIOS
- ETEQ's Cougar chipset
- 256 KB 2nd-level cache memory
- Up to 16 MB of on-board DRAM
- Built-in IDE (AT bus) hard disk drive interface
- Built-in floppy disk drive controller
- Two serial RS-232 ports
- One parallel/printer port
- On-board keyboard connector
- Lithium battery back-up for real-time clock/calendar

Specifications

- **CPU**: 80486SX/DX/DX2-25/33/40/50/66 MHz
- **Cache memory size**: 8 KB on-chip and 256 KB 2nd level
- **Bus interface**: ISA (PC/AT) bus
- **Data bus**: 32 bit
- **Processing ability**: 32 bit
- **Coprocessor**: Socket for Weitek 4167
- **RAM memory**: 1 MB to 64 MB, uses four banks of SIMM sockets composed of eight 30 pin sockets and two 72-pin sockets (72-pin sockets accept 1, 2, 4, 8 and 16 MB SIMMs)
- **Shadow RAM memory**: Supports system and video BIOS of up to 256 KB in 32 KB blocks
- **IDE hard disk drive interface**: Supports up to two IDE (AT bus) hard disk drives. BIOS enabled/disabled
- **Floppy disk drive interface**: Supports up to two floppy disk drives, 5.25" (360 KB and 1.2 MB) and/or 3.5" (720 KB and 1.44 MB). BIOS enabled/disabled
- **Bi-directional parallel port**: Configurable to LPT1, LPT2, LPT3 or disabled. Standard DB-25 female connector provided
- **Serial ports**: Two RS-232 serial ports can be individually set to COM1, COM2 or disabled. Each can be accessed through a DB-9 male connector
- **Real time clock/calendar**: Dallas DS-1287 with lithium battery back-up for 10 years of data retention
- **Watchdog timer**: Jumper configurable to, always disabled or software enabled/disabled. The timer interval is 1.6 sec. Your program uses I/O ports hex 043 and 443 to control the watchdog timer and generate a system reset or IRQ15
- **Piggyback connector**: 16-bit bus connector (64 + 36 pins) for expansion modules
- **DMA channels**: 7
- **Interrupt levels**: 15
- **Keyboard connector**: A 6-pin mini DIN keyboard connector is located on the mounting bracket for easy access. An external keyboard adapter is included. An on-board keyboard pin header connector is also available
- **Bus speed**: 8 MHz
- **System performance (w/ 80486DX-50 MHz CPU)**: 200 MHz, Landmark speed V1.14; 167 MHz, Landmark speed V2.0
- **Max. power requirements**: +5 V @ 2.5 A
- **Power supply voltage**: +5 V (4.75 V to 5.25 V), +12 V, -12 V
- **Operating temperature**: 32 to 140°F (0 to 60°C)
- **Board size**: 13.1" (L) x 4.8" (W) (334 mm x 122 mm)
- **Board weight**: 1.2 lbs (0.5 Kg)

Ordering Information

- ❏ **PCA-6147-33/Bare**: All-in-one 80486 CPU Card without CPU. Includes 256 KB cache memory, users manual, IDE hard disk cable, floppy drive cable and parallel port adapter
- ❏ **PCA-6147SX-25/0K**: Same as above but with 25 MHz 80486SX CPU installed
- ❏ **PCA-6147SX-33/0K**: Same as above but with 33 MHz 80486SX CPU installed
- ❏ **PCA-6147DX-33/0K**: Same as above but with 33 MHz 80486DX CPU installed
- ❏ **PCA-6147DX-50/0K**: Same as above but with 50 MHz 80486DX CPU installed
- ❏ **PCA-6147DX2-50/0K**: Same as above but with 50 MHz 80486DX2 CPU installed
- ❏ **PCA-6147DX2-66/0K**: Same as above but with 66 MHz 80486DX2 CPU installed

On-board POST diagnostic LEDs
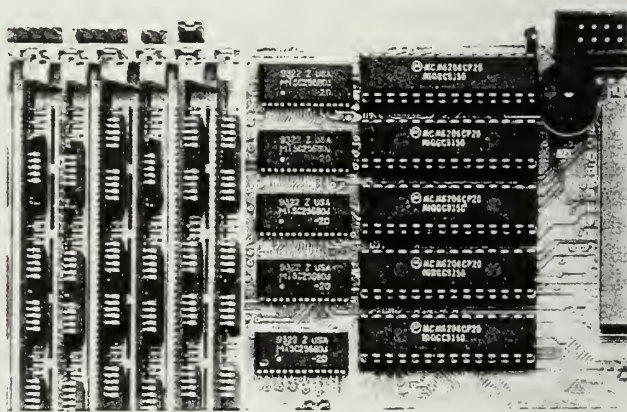
Specifications

CPU: 80486SX/DX/DX2-25/33/50/66 MHz

Cache memory size: 256 KB

Bus interface: ISA (PC/AT) bus

Data bus: 32 bit

Processing ability: 32 bit

Chipset: ETEQ's Cougar chipset

RAM memory: 1 MB, 4 MB and 16 MB. Uses 256Kx9 (SIMM-256-8), 1Mx9 (SIMM-1000-8) or 4Mx9 (SIMM-4000-8) SIMMs with access time of 80 ns or less

CPU Comparison:

CPU	80486DX-33	80486DX2-50	80486DX2-66
Co-processor	Built-in	Built-in	Built-in
Cache memory	8 KB + 256 KB	8 KB + 256 KB	8 KB + 256 KB
Landmark Speed 1.14	150.2 MHz	170.1 MHz	>200 MHz
System Clock	33 MHz	25 MHz	33 MHz

Shadow RAM memory: Supports up to 256 KB of memory in 16 KB blocks for system and video BIOS

Hard disk drive interface: Supports up to two IDE (AT-Bus) hard disk drives. Jumper enabled/disabled

Floppy disk drive interface: Supports up to two floppy disk drives: 5¼" (360 KB and 1.2 MB) and/or 3½" (720 KB and 1.44 MB). Jumper enabled/disabled

Parallel/printer port:
Configurable to LPT1, LPT2, LPT3 or disabled. A standard female DB-25 connector is provided

Serial ports: Two RS-232 serial ports individually configurable to COM1, COM2 or disabled. Each port is accessed through its own male DB-9 connector

Real time clock/calendar:
Real time clock/calendar with lithium battery back-up (3.6 V @ 850 mAH). External battery connector provided

Watchdog timer: Jumper configurable to always ON, always OFF, or programmable ON/OFF. The time-out interval is jumper selectable to 1.5, 15 or 150 seconds

Piggyback connector: 64-pin, 8-bit bus connector with a low-line detector and battery back-up reserved for option modules such as Flash/RAM/ROM disk module and/or Flat-panel/CRT VGA modules

DMA channels: 7

- **Interrupt levels**: 15
- **Keyboard connectors**: A 6-pin mini-DIN keyboard connector is located on the mounting bracket for easy access. An external keyboard adapter is included. An on-board keyboard pin header connector is also available
- **Bus speed**: 8 MHz.
- **System performance**: 150 MHz with an 80486DX-33 MHz (Landmark speed V1.14).
- **Max. power requirements**: +5 V @ 2.5 A
- **Operating temperature**: 32 to 140°F (0 to 60°C)
- **Board size**: 13.1" (L) x 4.8" (W) (334 mm x 122 mm)
- **Board weight**: 1.5 lbs (0.7 Kg)
- **EMI**: meets FCC class A and BZT Class A
- **MTBF**: 87,100 hrs @ 25°C; 31,900 hrs @ 60°C

Ordering Information

❑ **PCA-6146-33/Bare**:
All-in-one 80486 CPU Card without CPU. Includes 256 KB memory, user's manual, IDE hard disk drive cable, floppy disk drive cable, parallel port adapter and keyboard adapter.

❑ **PCA-6146SX-25/0K**:
All-in-One 80486SX-25 CPU Card with 256 KB cache memory and all accessories of the PCA-6146-33/Bare

❑ **PCA-6146SX-33/0K**:
All-in-One 80486SX-33 CPU Card with 256 KB cache memory and all accessories of the PCA-6146-33/Bare

❑ **PCA-6146DX-33/0K**:
All-in-One 80486DX-33 CPU Card with 256 KB cache memory and all accessories of the PCA-6146-33/Bare

❑ **PCA-6146DX2-50/0K**:
All-in-one 80486DX2-50 CPU Card with 256 KB cache memory and all accessories of the PCA-6146-33/Bare
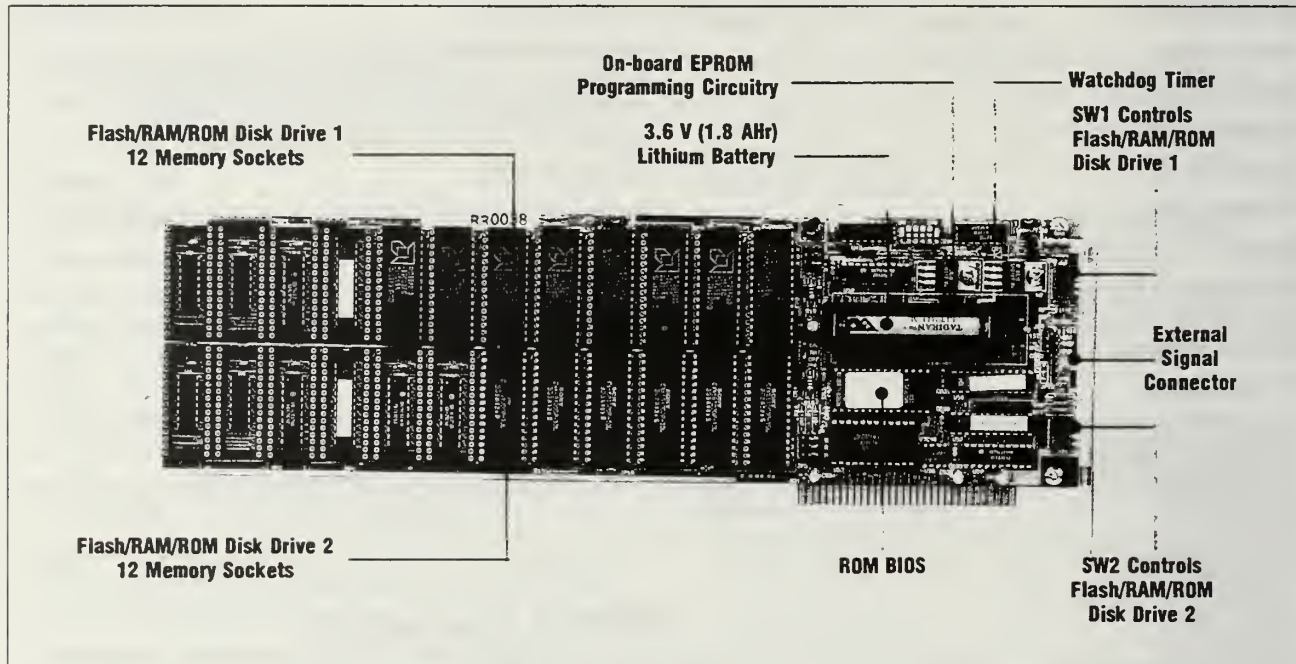
❑ **PCA-6146DX2-66/0K**:
All-in-one 80486DX2-66 CPU Card with 256 KB cache memory and all accessories of the PCA-6146-33/Bare

ISBC Series

PCD-890 Dual Flash/RAM/ROM Disk Card

Flash/RAM/ROM Disk Drive 1
12 Memory Sockets

On-board EPROM Programming Circuitry

3.6 V (1.8 AHr) Lithium Battery

Watchdog Timer

SW1 Controls Flash/RAM/ROM Disk Drive 1

External Signal Connector

Flash/RAM/ROM Disk Drive 2
12 Memory Sockets

ROM BIOS

SW2 Controls Flash/RAM/ROM Disk Drive 2

Introduction

The PCD-890 solid-state disk emulates two floppy disk drives. It provides anywhere from 360 KB to 12 MB of storage using Flash/EPROM/SRAM memories. When you replace mechanical disk drives with the PCD-890, your critical PC applications will run faster in harsh industrial environments with a higher degree of reliability.

The size of the PCL-890's disks depends on the number of chips installed. The unit works with a wide assortment of supported chips from standard manufacturers or their equivalents. You can designate each as drive A, B, C or D. You can install up to two PCD-890s in your PC for at total of 24 MB of storage.

The PCD-890's on-board watchdog timer protects your applications from system standstills, particularly useful in stand-alone or unattended environments requiring auto reset or auto reboot.

Applications

- Diskless PCs
- High-reliability industrial PCs
- Stand-alone or unmanned machines
- Sites that demand high-speed or heavy-duty disk operations
- Industrial controllers
- Network terminals
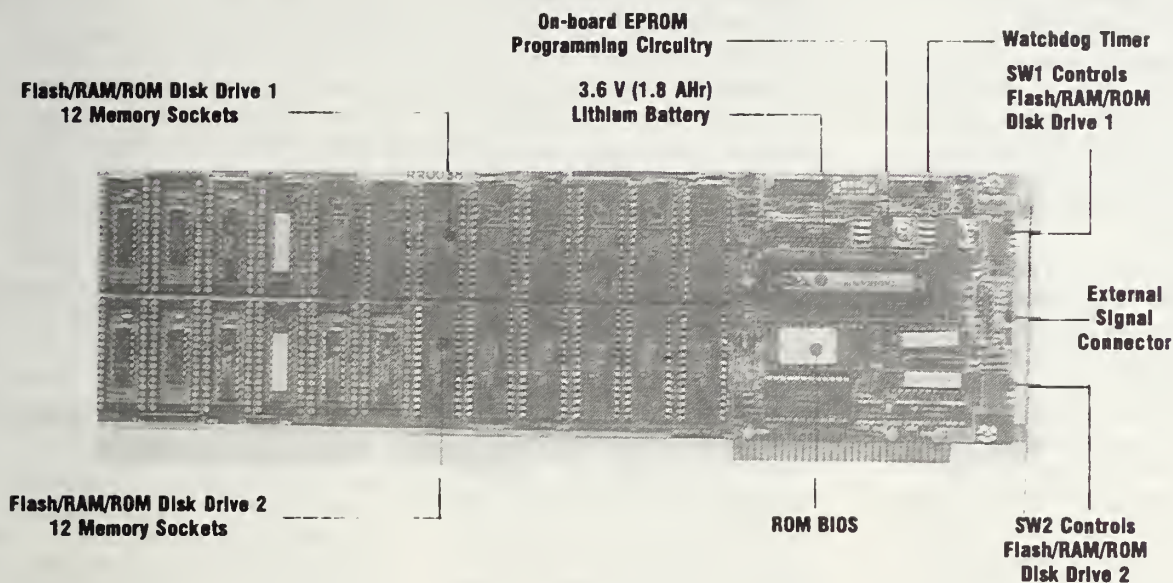- Industrial PCs requiring high-speed disk I/O

Features

- Emulates up to two floppy disk drives
- Disk sizes: 360 KB to 12 MB (both banks linked together)
- Drive designation: DOS drive A, B, C or D (1st, 2nd, 3rd and 4th FDDs)
- Offers 24 individual 32-pin memory sockets divided into two banks, one bank for each drive
- Accepts 128Kx8 Flash/EPROM/SRAM or 512Kx8 Flash/EPROM/SRAM
- Fully software-compatible with mechanical floppy disk drives. Requires no special software development

- Power-on auto-boot feature; user-defined password and user's prompt, excellent for OEMs
- Up to two PCD-890s can be installed in one PC
- On-board EPROM programming circuitry with easy-to-use menu driven programming utility software
- Lithium backup battery (3.6 V @ 1.8 AHr) for 5-year data retention (with maximum load of 24 SRAM chips)
- Connector for external battery
- Each card occupies only 16 KB of system memory space
- Watchdog timer with selectable time-out period (100 msec and 1.6 sec)
- Memory-mapped data transfer
- Switch (enable/disable) between floppy disk drives and PCD-890s by software
- Connector with pins for +5 V, +12 V, GND, PFO (Power Failure Output) and WDO (Watchdog Output) signals
- All solid-state construction for environments hostile to diskettes

PCD-890 Dual Flash/RAM/ROM Disk Card

Flash/RAM/ROM Disk Drive 1
12 Memory Sockets

On-board EPROM
Programming Circuitry

3.6 V (1.8 AHr)
Lithium Battery

Watchdog Timer

SW1 Controls
Flash/RAM/ROM
Disk Drive 1

External
Signal
Connector

Flash/RAM/ROM Disk Drive 2
12 Memory Sockets

ROM BIOS

SW2 Controls
Flash/RAM/ROM
Disk Drive 2

Introduction

The PCD-890 solid-state disk emulates two floppy disk drives. It provides anywhere from 360 KB to 12 MB of storage using Flash/EPROM/SRAM memories. When you replace mechanical disk drives with the PCD-890, your critical PC applications will run faster in harsh industrial environments with a higher degree of reliability.

The size of the PCL-890's disks depends on the number of chips installed. The unit works with a wide assortment of supported chips from standard manufacturers or their equivalents. You can designate each as drive A, B, C or D. You can install up to two PCD-890s in your PC for at total of 24 MB of storage.

The PCD-890's on-board watchdog timer protects your applications from system standstills, particularly useful in stand-alone or unattended environments requiring auto reset or auto reboot.

Applications

- Diskless PCs
- High-reliability industrial PCs
- Stand-alone or unmanned machines
- Sites that demand high-speed or heavy-duty disk operations
- Industrial controllers
- Network terminals
- Industrial PCs requiring high-speed disk I/O

Features

- Emulates up to two floppy disk drives
- Disk sizes: 360 KB to 12 MB (both banks linked together)
- Drive designation: DOS drive A, B, C or D (1st, 2nd, 3rd and 4th FDDs)
- Offers 24 individual 32-pin memory sockets divided into two banks, one bank for each drive
- Accepts 128Kx8 Flash/EPROM/SRAM or 512Kx8 Flash/EPROM/SRAM
- Fully software-compatible with mechanical floppy disk drives. Requires no special software development

- Power-on auto-boot feature; user-defined password and user's prompt, excellent for OEMs
- Up to two PCD-890s can be installed in one PC
- On-board EPROM programming circuitry with easy-to-use menu driven programming utility software
- Lithium backup battery (3.6 V @ 1.8 AHr) for 5-year data retention (with maximum load of 24 SRAM chips)
- Connector for external battery
- Each card occupies only 16 KB of system memory space
- Watchdog timer with selectable time-out period (100 msec and 1.6 sec)
- Memory-mapped data transfer
- Switch (enable/disable) between floppy disk drives and PCD-890s by software
- Connector with pins for +5 V, +12 V, GND, PFO (Power Failure Output) and WDO (Watchdog Output) signals
- All solid-state construction for environments hostile to diskettes

Q&A *Solid State Storage Devices*

Q. What special features does a Flash/RAM/ROM disk offer for the industrial environment?

A. Flash/RAM/ROM disks are the solid-state equivalents of mechanical disk-drives. They offer faster data access and longer MTBF, characteristics which make them the ideal solution for critical commercial or industrial applications.

Mechanical disks are highly susceptible to breakdown in severe industrial environments. A Flash/RAM/ROM disk uses Flash, SRAM and EPROM memory to store your data and application programs instead of the magnetic particles on a rotating disk. Although the initial cost for the solid-state disk is higher, it gives you faster and more efficient operation, a longer lifespan and a lower risk of breakdown or data loss during critical manufacturing or commercial processes.

Q. What is a memory-card drive?

A. A memory-card drive uses credit-card sized memory cartridges to store data using procedures established in the PCMCIA 1.0/JEIDA 4.0 standard. Card drives link with the PC/ISA bus and allow you to write to and read from an IC memory card as you would a magnetic disk. Like a floppy disk, you can remove a cartridge from a drive on one PC and use it in another PC's card drive.

Our memory-card drives offer a seek time that is orders of magnitude faster than mechanical disks. Card drives are also much less vulnerable to wear, part failure or vibration. Industrial PCs are only one of many candidates for these systems. Memory card drives are ideal in any environment that requires portability, ruggedness and fast access. That's why they are popular for fleet vehicles, robots, remote data loggers and mobile computer systems.

Q. What are the differences between applications for Flash/RAM/ROM disk cards and applications for memory-card drives?

A. Flash/RAM/ROM disks appear in applications which demand large storage capacity, easy memory expansion, complete DOS compatibility and the security which diskless operation provides.

They make excellent direct replacements for mechanical drives because they completely emulate DOS operations, withstand more severe conditions and read and write much faster. Their watchdog timers make stand-alone or unmanned operations much easier to manage because they can trigger auto-resets or auto-reboots in case of power failures. Disk cards also work well in high-security environments because they're entirely enclosed within your PC and therefore far more tamper-proof than disk drives. Applications that generate lots of data will find ample storage space on a disk card.

Any application that rewards portability, mobility and low power consumption will benefit from a memory-card drive. They've won favor with designers of test equipment, data-control systems and data loggers because of their small size, light weight and the availability of standard memory cards in several sizes from 128 KB to 64 MB. The cards themselves weigh little (from 1 to 1.5 ounces [138 to 206 g]) and can be moved from drive to drive just like floppy disks.

Q. What are the different types of memories used in solid-state disks?

A. Three types of memory are available: EPROMs, battery-backed SRAM and Flash memory. All three types offer you storage capacities that equal or beat those of floppy disks. At the same time, since they have no moving parts, they offer greater reliability than mechanical drives.

An **EPROM** (Erasable Programmable Read-Only Memory) provides storage that is nearly nonvolatile, for it is written electrically and can only be erased by UV light. **SRAMs** with battery backing are normal static RAMs coupled with a battery that retains data when normal power is withdrawn. **Flash memory** operates like an EPROM, except that it can be programmed and erased while on-board. It provides the same long data retention but reduces the time required to store the data.

Q: How does the solid state disk work?

A: The solid state disk uses memory chips (Flash, SRAM or EPROM) to store programs and data instead of the magnetic particles on the mechanical drive's disk. When the system boots, the disk card modifies the BIOS INT-13 disk I/O routine. The routine then translates read and write commands to the disk card so that they will correctly access the memory chips. You don't need any special drivers. You simply set the drive to act as drive A: or C: and use standard DOS commands (COPY, DIR, etc.) to manipulate your data.

If you use Flash or SRAM for the solid state disk, you can read or write data. If you use EPROM, files on the disk are read only. The PCD-890 can program some common EPROM chips on board. Otherwise you will need an external programmer to load your program and data files on the EPROMs.

Q: How do I boot from a solid state disk?

A: It's easy. Simply set the jumpers on your solid state disk to emulate drive A: (the 1st FDD), then copy your application files to the disk along with the standard system files required to boot (command.com, io.sys, autoexec.bat, etc). Next time you start your computer, it will boot from the solid state disk.

Specifications

Flash: ATMEL 29C010 (128 Kx8),
29C040 (512 Kx8)
INTEL or AMD 28F010 (128 Kx8)

EPROM: ATMEL 27C010 (128 Kx8),
27C040 (512 Kx8)

SRAM: CXK581000P (128 Kx8),
CXK584000P (512 Kx8)

Note: You may use code-equivalent
chips but make sure to use only
memories from recognized suppliers

Battery: 3.6 V (1.8 AHr) lithium
battery backup

Operating temperature:
32 to 140°F (0 to 60°C)

Power: +5 V @ 1 A maximum for
normal applications, +12 V @ 300 mA
maximum for programming EPROMs

Board size: 13.3" (L) x 4.2" (W)
(340 mm x 107 mm)

Ordering Information

❑ **PCD-890**:
Dual Flash/RAM/ROM Disk Card with
0 KB memory, user's manual and
utility diskette

❑ **M-27C010x3**:
Three 128 KB EPROM devices

❑ **M-27C040x3**:
Three 512 KB EPROM devices

❑ **M-581000x3**:
Three 128 KB SRAM devices

❑ **M-584000x3**:
Three 512 KB SRAM devices

❑ **M-29C010x3**:
Three 128 KB (+5 V) Flash memories

❑ **M-29C040x3**:
Three 512 KB (+5 V) Flash memories

Memory Configuration

The following table shows the number of EPROM, Flash or SRAM chips required for
each disk size.

Device		360 KB	720 KB	1.2 MB	1.44 MB	2.88 MB	6 MB
EPROM	27C010	3	6	10	12	24	-
	27C040	1	2	3	3	6	12
SRAM	58256	12	-	-	-	-	-
	581000	3	6	10	12	24	-
	584000	1	2	3	3	6	12
Flash	29C010	3	6	10	12	24	-
	28F010	3	6	10	12	24	-
	29C040	1	2	3	3	6	12

*IPC
Peripherals*

PCD-892 *Flash/RAM/ROM Disk Card*
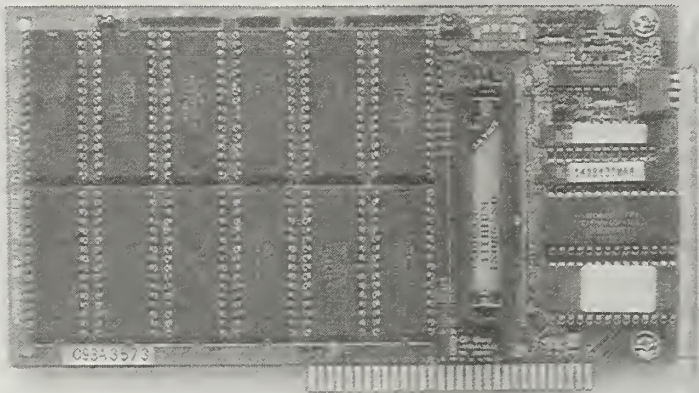
Introduction

The PCD-892 half-size Flash/RAM/ROM disk card uses up to 6 MB of SRAM, EPROM or Flash memory chips to replace a floppy disk drive. It offers faster access times and better protection from the vibration, vapors and contaminants found in harsh industrial environments. The emulated drive is identified conventionally as A, B, C or D and obeys standard DOS commands; no special software is required.

An optional lithium battery (3.6 V, 1.8 AHr) preserves data stored on an SRAM disk in case of power failure. The PCD-892 also comes equipped with a watchdog timer which outputs a TTL-low signal if the CPU's processing comes to a halt due to a software bug or EMI. You can use this signal to activate an LED or alarm or to trigger an auto-reset or auto-reboot.

Applications

- Programs that require frequent, high-speed disk access
- Diskless PCs and workstations
- Security systems
- Embedded control systems
- Unmanned (run-only) controllers
- Industrial control systems
- Instrumentation systems
- Testing systems

Features

- PC/AT compatible half-size card
- Can be enabled or disabled in software
- Fully software-compatible with conventional drives, requires no special software development
- Auto-bootable when emulating drive A
- Disk size from 360 KB to 6 MB
- Accepts 128Kx8 Flash/EPROM/SRAM or 512Kx8 Flash/EPROM/SRAM
- Lithium battery (3.6 V @ 1.8 AHr) for SRAM data retention of no less than ten years
- On-board connections for external battery, V_{cc} and +12 V power sources, power failure warning and watchdog timer outputs
- Each card occupies only 16 KB of system memory space
- Selectable watchdog timer intervals of 100 msec and 1.6 sec
- Memory-mapped data-transfer
- 32 to 140°F (0 to 60°C) operating temperature
- Password protection against unauthorized changes
- User-defined prompt offers easy customizing for OEMs

Specifications

- **Supports the following memory devices**:

  Flash
  ATMEL 29C010 +5 V (128 Kx8), ATMEL 29C040 +5 V (512 Kx8) AMD/INTEL 28F010 +12 V (128 Kx8) or equivalent. Approved manufacturers only.

  EPROM
  ATMEL 27C010 (128 Kx8), 27C040 (512 Kx8) or equivalent. Approved manufacturers only.

  SRAM
  CXK 581000P (128 Kx8), CXK584000P (512 Kx8) or equivalent. Approved manufacturers only.

- **Power requirements**:
  +5 V @ 0.5 A max. (normal operations); +12 V @ 50 mA max. (during flash memory programming)

- **Board size**:
  7.3" x 3.9" (185 mm x 98 mm)

Ordering Information

❑ **PCD-892A**:
Flash/RAM/ROM Disk Card with battery

❑ **PCD-892B**:
Flash/ROM Disk Card without battery

❑ **M-27C010x3**:
Three 128 KB EPROM devices

❑ **M-27C040x3**:
Three 512 KB EPROM devices

❑ **M-581000x3**:
Three 128 KB SRAM devices

❑ **M-584000x3**:
Three 512 KB SRAM devices

❑ **M-29C010x3**:
Three 128 KB (+5 V) Flash memories

❑ **M-29C040x3**:
Three 512 KB (+5 V) Flash memories

PCL-844 8-port Intelligent RS-232 Card

Features

- 12 MHz 80286 processor, CD180 RISC-Based UART, 512 KB dual ported RAM.
- Baud rate up to 38400 bps with eight ports on line
- Complete RS-232 modem control signals
- Maps to just 16 KB of system memory. Choose one of six addresses from C8000 to DC000.
- Many IRQ options: 2, 3, 4, 5, 7, 10, 11, 12 or 15
- Easy-to-use menu driven installation program
- LEDs on connection box let you monitor the TxD/RxD status of any port
- Links to peripherals up to 4000 ft from controller (RS-422)

Applications

- Data acquisition and control with RS-232/RS-422 based devices
- PLC monitoring and control
- Instrument controller, distributed control system
- Modem server, database server, POS controller
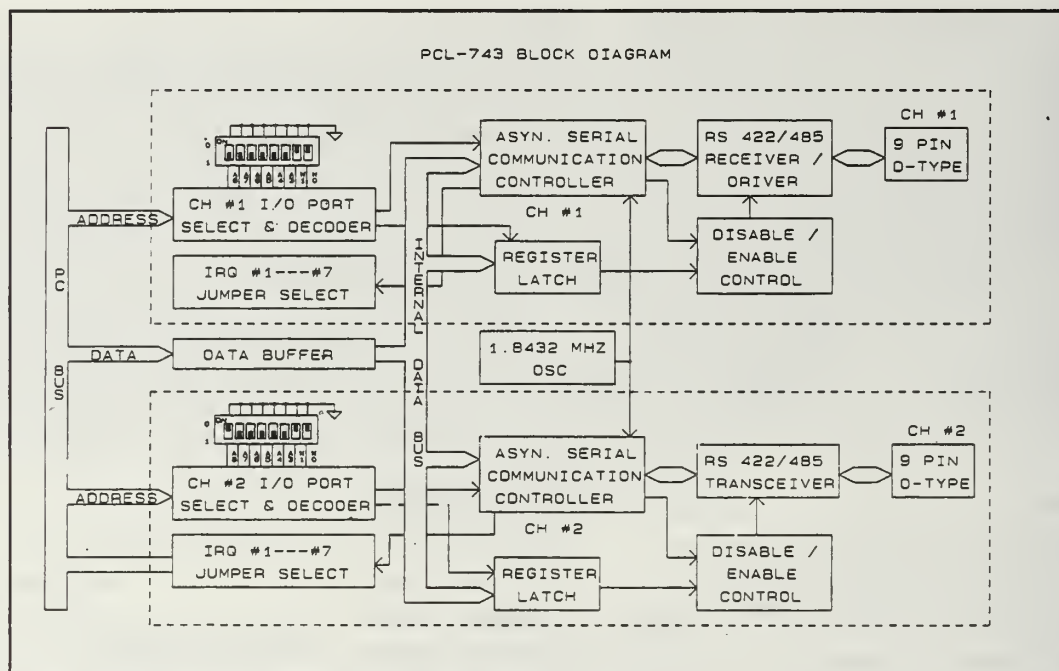- Multi-user system

Specifications

Board

- **Number of ports**: 8
- **Processor**: 12 MHz 80286
- **Dual-ported RAM**: 512 KB
- **SRAM**: 16 KB
- **UART**: RISC-based CD180
- **Total ports in one system**: 32
- **Operating temperature**: 32 to 122°F (0-50°C)
- **Power consumption**: +5 V @ 1.5 A, +12 V @ 120 mA, -12 V @ 180 mA
- **Dimensions**: 13.3 x 4.7 in. (338 x 120 mm)
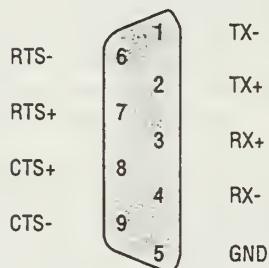- **Weight**: 1.5 lb (0.67 Kg)

RS-232 interface

- **Signals**: TxD, RxD, RTS, CTS, DTR, DSR, DCD and GND
- **Mode**: asynchronous full duplex
- **Communication rate**: 50 to 38,400 bps
- **Stop bits**: 1 or 2
- **Parity**: even, odd or none
- **Data bits**: 7 or 8

Introduction

We designed the PCL-844 Intelligent 8-port RS-232 or RS-422 interface card for lab and industrial applications where a PC needs to communicate with terminals, modems or other instruments. RS-422 applications use the optional PCL-8442 8-port isolated RS-232 to RS-422 converter, shown on the following page. You can install up to four PCL-844 cards for a total of 32 ports in any AT/ISA bus 286/386/486 based PC.

The PCL-844's on-board 12 MHz 80286 processor takes over the communications load from the host PC. When you are processing large amounts of data from multiple ports, servicing the interrupts alone consumes a large percentage of the capacity of your computer's CPU. The PCL-844 serves as a high-speed dedicated interrupt processor. Its CPU directly controls the board's CD180 RISC-based UART, guaranteeing 38,400 bps performance over eight high-speed data ports.

The PCL-844 is virtually a self contained computer in its own right. It contains 512 KB of dual-ported RAM which you can use to store and run programs. The dual-port RAM maps into the host system's address space to give you the fastest possible data transfers between the PCL-844 and PC-memory.

When the PCL-844 initializes, it downloads the driver software (which functions like a PC's BIOS) into on-board SRAM. This improves performance and makes version upgrading easy, with no hardware redundancy.

Each PCL-844 comes with software drivers for DOS and Windows (PC-ComLIB, described on the following page). These drivers support most common languages, including C, Pascal, Visual Basic, Quick Basic, assembly and Clipper. The PC-ComLIB package also includes the DataScope data viewer, terminal emulator and self-diagnostic utilities for easy troubleshooting and debugging.

RS-422/485 Interface Card Block Diagram

Pin Assignment

RTS-	6	1 TX-
RTS+	7	2 TX+
CTS+	8	3 RX+
CTS-	9	4 RX-
		5 GND

Wiring Diagram (2-wire)

## RS-485 Programming Example

```
10      'Configured as COM1 with the
        driver/receiver bit enabled
20      BASE% = &3F8
...
100     OPEN "COM1:9600,N,8,1,RS" AS #1
110     OUT BASE%+7,1       'Enable driver
120     PRINT #1, DATA1$     'Send data
...
200     OUT BASE%+7,2       'Enable receiver
210     INPUT #1, DATA2$    'Receive data
...
300     OUT BASE%+7,0       'Disable driver
```

Ordering Information

❑ **PCL-743:**
   General-purpose RS-422/485 Interface Card, user's manual

❑ **PCL-745:**
   Isolated RS-422/RS-485 Interface Card, user's manual

❑ **PCLS-802:**
   PC-ComLIB Serial Communication Programming Library

PCL-812/812PG MultiLab Analog & Digital I/O Card
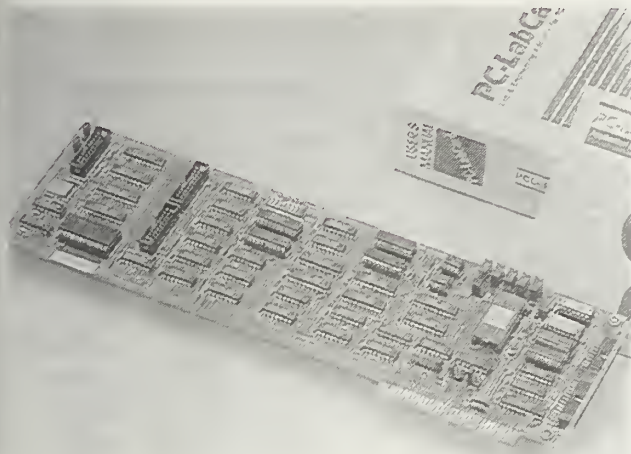
Introduction

The PCL-812 and PCL-812PG are multifunction analog and digital I/O cards which offer the five most desired measurement and control functions for PC/AT and compatible systems: A/D conversion, D/A conversion, digital input, digital output and counter/timer. They neatly package 16 12-bit analog input channels, two 12-bit analog output channels, 16 digital input channels, 16 digital output channels and a programmable counter/timer on a full-size card.

In addition to all the features listed above the PCL-812PG offers the convenience of programmable analog input ranges. With the PCL-812PG selection of an analog input range is not done by DIP switches, but by software commands. For applications which need different gains for different channels or different gains for different stages of a process, the PCL-812PG offers convenience and maximum resolution.

Rich software support, numerous I/O options and a wide range of available daughterboards make the PCL-812 and PCL-812PG ideal for industrial applications that require a combination of analog and digital I/O.

Features

PCL-812 and PCL-812PG

- 16 single-ended 12-bit analog input channels
- Two 12-bit analog output channels
- Programmable sampling rate of up to 30 KHz
- A/D with DMA or interrupt
- 16 digital output channels
- 16 digital input channels
- Programmable timer/counter

- Includes C/C++, PASCAL and BASIC drivers as well as calibration, demo and example programs
- Rich application software support
- Wide variety of external daughter boards

PCL-812PG only

- Programmable A/D ranges (gains)

Applications

- DC voltage measurement
- Transducer/sensor interfacing
- Waveform analysis
- Process control
- Programmable voltage output
- Contact closure monitoring
- Digital signal and BCD interfacing
- Industrial ON/OFF control
- Multiplexer and relay control
- Frequency, period and pulse width measurement
- Event counting and pulse train generation

I/O functions

Analog input

The PCL-812 and PCL-812PG use an industrial standard 12-bit successive approximation A/D converter (AD574) with sample and hold for accurate, high-speed A/D conversions. The typical conversion time is 25 microseconds.

You can trigger the A/D conversion in three ways: by program control, by on-board programmable pacer or by an external trigger pulse. The on-board pacer uses two 16-bit timer/counter channels from an Intel 8253. A crystal oscillator provides a 2 MHz time base. This oscillator lets the pacer generate trigger pulses with frequencies ranging from 500 KHz to 0.00046 Hz (1 pulse every 36 minutes).

You can perform A/D data transfer in three ways: by program control, by interrupt service routine or by DMA. If you use interrupt data transfer you can jumper-select any IRQ level between 2 and 7. If you use DMA data transfer you can jumper select either DMA channel 1 or 3.

Analog output

As a complement to the analog inputs, the PCL-812 and PCL-812PG also provide two 12-bit double-buffered analog output channels. You can operate their D/A converters with an internal fixed reference in the 0 to 5 V output range or with an external reference for 0 to +10 V or 0 to -10 V output.

Block diagram for the PCL-1800

Pin Assignment
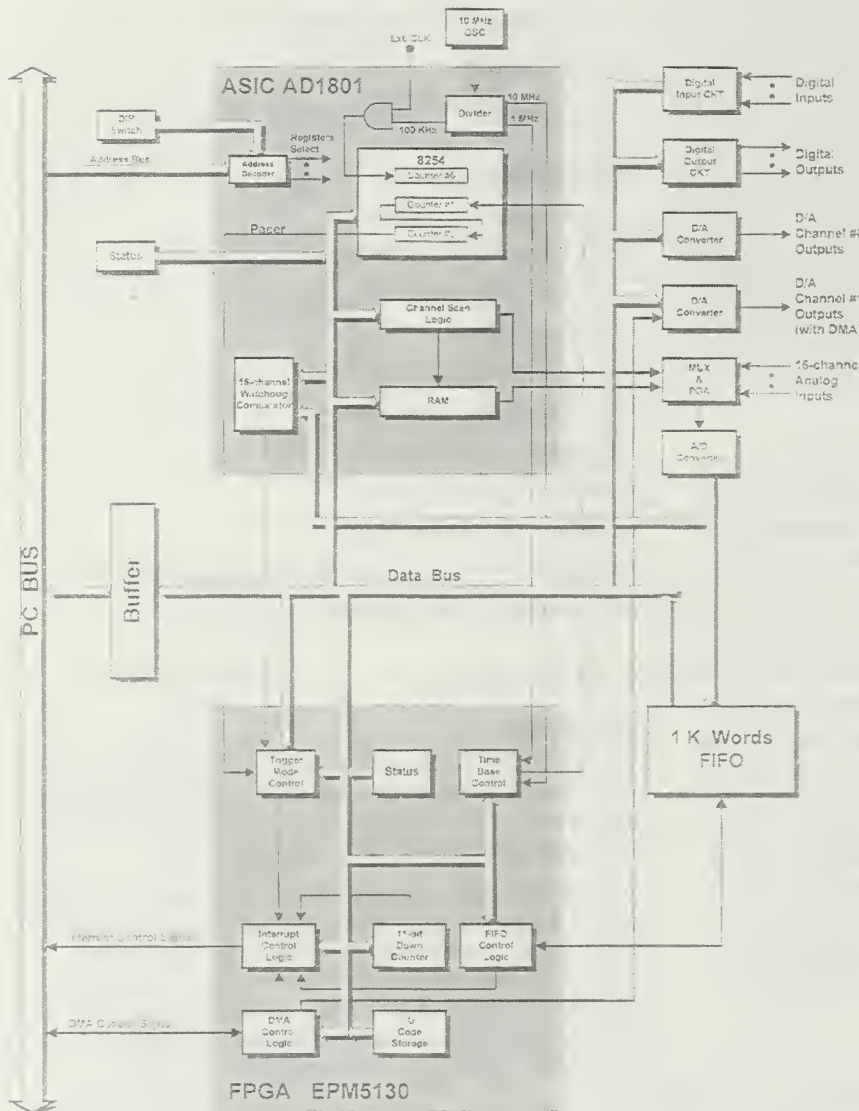
Connector 1 (A/D, D/A)

+12 V	1		
AIH0	2	20	AGND
AIH1	3	21	AIL8
AIH2	4	22	AIL9
AIH3	5	23	AIL10
AIH4	6	24	AIL11
AIH5	7	25	AIL13
AIH6	8	26	AIL14
AIH7	9	27	AIL15
AGND	10	28	AIL15
VREF	11	29	AGND
AGND	12	30	DA0OUT
DA1VREF	13	31	DA0VREF
AGND	14	32	DA1OUT
DGND	15	33	AGND
DAECLK	16	34	DGND
ECLK	17	35	TRIG0
OUT0	18	36	GATE0
+5 V	19	37	OUT2

Connector 2 (D/O)

D/O 0	1	2	D/O 1
D/O 2	3	4	D/O 3
D/O 4	5	6	D/O 5
D/O 6	7	8	D/O 7
D/O 8	9	10	D/O 9
D/O 10	11	12	D/O 11
D/O 12	13	14	D/O 13
D/O 14	15	16	D/O 15
D.GND	17	18	D/O D.GND
+5 V	19	20	+ 12 V

Connector 3 (D/I)

D/I 0	1	2	D/I 1
D/I 2	3	4	D/I 3
D/I 4	5	6	D/I 5
D/I 6	7	8	D/I 7
D/I 8	9	10	D/I 9
D/I 10	11	12	D/I 11
D/I 12	13	14	D/I 13
D/I 14	15	16	D/I 15
D.GND	17	18	DN D.GND
+5 V	19	20	+12 V

Ordering Information

□ PCL-1800:
330 KHz High-speed DAS Card, users manual and utility diskette with BASIC,
C, C++ and Pascal drivers

tal I/O

PCL-812 and PCL-812PG come with 16 digital inputs and
igital outputs, accessed via two 20-pin dual-in-line
nectors. These connectors are standard on most I/O cards
daughterboards in the PC-LabCard family. Digital inputs
normally set high (value = 1) without any input and
ge state with the input signals accordingly. Digital
uts are normally set low (value = 0) at initial state and
at the same state (buffered) until the next output
ation occurs.

er/counter

third timer/counter channel on the Intel 8253, powered by
ternal or external time base, can be used to count events
easure frequency, period and pulse width.

t state insertion

use of the wide variety of CPU and bus speeds in the
ket we designed the PCL-812 and 812PG with a wait-state
tion capability. Wait-state insertion addresses most
d compatibility problems, allowing you to use these cards
Cs with speeds ranging from 16 MHz (80286) up to 66
(80486DX2). This feature ensures that your card will
up with future technology.

ecifications

log input

annels: 16 single-ended

solution: 12 bits

nverter: Honeywell HADC-574ACCJ or equivalent

nversion time: 25 microseconds (max. 30 KHz)

ut range (in V):
L-812:   ±10, ±5, ±2, ±1
L-812PG: ±10, ±5, ±2.5, ±1.25, ±0.625, ±0.3125

nge selection:
L-812 by DIP switches, PCL-812PG by software.

gger mode: by software, on-board/external trigger

ta transfer: by program control, interrupt (IRQ 2 to 7) or
MA (Channel 1 or 3) for single channel scan

curacy: 0.01% of reading ±1 bit

mmon mode rejection: 60 dB typical

ut impedance: >10 MΩ

ervoltage: Continuous ±30 V_{DC} max.

Analog output

- **Channels**: Two double-buffered 12-bit channels

- **D/A range (in V)**:
  PCL-812: 0~5, 0~10 (w/external reference)
  PCL-812PG: 0~5, 0~10 (w/internal reference); ±10 V max.
  with external AC or DC reference (accuracy for output above
  ±9 V may vary depending on power supply used)

- **Settling time**: 30 microseconds

- **Output current**: ±10 mA max.

- **D/A device**: MP7623KN or AD7541AKN or equivalent

- **Linearity**: ±½ bit

Digital input

- **Channels**: 16

- **Logic level 0**: 0 to 0.8 V_{DC}

- **Logic level 1**: 2.0 to 5.0 V_{DC}

- **Input load**:
  0.4 mA max. @ 0.5 V (low), 50 µA max. @ 2.7 V (high)

Digital output

- **Channels**: 16

- **Logic level 0**: 0 to 0.4 V_{DC}

- **Logic level 1** : 2.4 to 5.0 V_{DC}

- **Driving capacity**: Sink: 8.0 mA @ 0.5 V
  Source: 0.4 mA @ 2.4 V

A/D pacer and counter (8253)

- **A/D pacer**: 32-bit timer with a 2 MHz time base.

- **Max. and min. rates**: 500 KHz to 0.00046 Hz (one sample
  every 36 minutes)

- **Counter**: One 16-bit counter with a 2 MHz time base

General

- **Power consumption**:
  + 5 V @ 500 mA typical, 1.0 A max.
  + 12 V @ 50 mA typical, 100 mA max.
  -12 V @ 14 mA typical, 20 mA max.

- **Operating temperature**: 32°F to 140°F (0 to 50°C)

- **I/O ports**: 16 consecutive bytes

- **Connectors**: All I/O channels are accessed through five on-
 board, 20-pin, dual-in-line connectors

- **Base address**:
 DIP-switch selectable, default setting is H220

PCL-812/812PG

Ordering Information

❏ **PCL-812**: PCL-812 Multi-Lab Card, user's manual and utility diskette, with BASIC, C/C++ and PASCAL drivers

❏ **PCL-812PG**: PCL-812PG, user's manual and utility diskette with BASIC, C/C++ and PASCAL drivers

❏ **OPT 002**: Wiring kit: includes PCLD-780 wiring terminal board, PCL-10501/PCL-10502 industrial wiring adapters and cables.

❏ **OPT 003**: Three application software packages: PCLS-700-1 PC-LabDAS, PCLS-800 PC-Scope and PCLS-702 LABTECH ACQUIRE

❏ **PCL-812-CS**: Complete package: PCL-812 + OPT 002 + OPT 003

❏ **PCL-812PG-CS**: Complete package: PCL-812PG + OPT 002 + OPT 003

❏ **PCLS-DLL-2**: Windows DLL driver for PCL-812/PG or PCL-711B
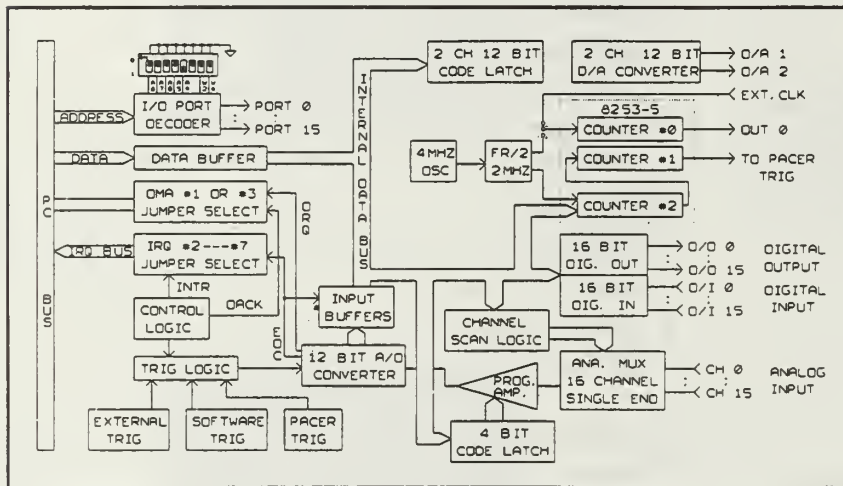
PCL-812PG Block Diagram
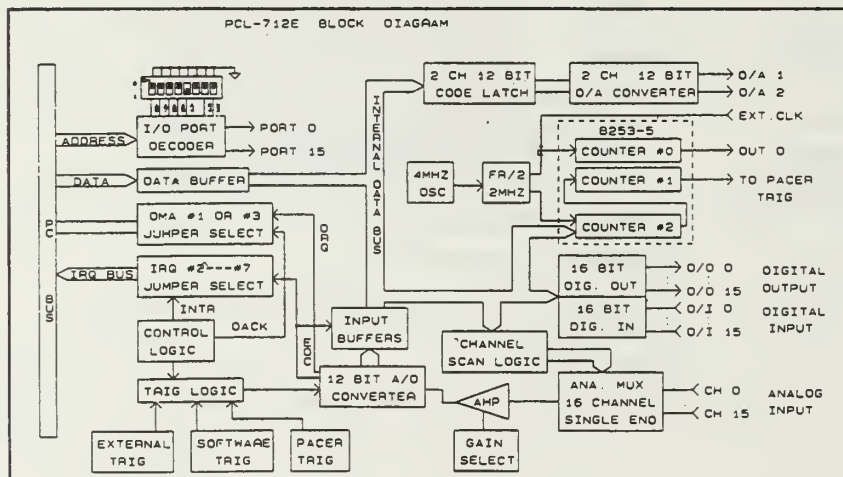
PCL-812 Block Diagram

Pin Assignment

Connector 1 (A/D)

A/D 0	1	2	A.GND
A/D 1	3	4	A.GND
A/D 2	5	6	A.GND
A/D 3	7	8	A.GND
A/D 4	9	10	A.GND
A/D 5	11	12	A.GND
A/D 6	13	14	A.GND
A/D 7	15	16	A.GND
A/D 8	17	18	A.GND
A/D 9	19	20	A.GND

Connector 2 (A/D, D/A)

A/D 10	1	2	A.GND
A/D 11	3	4	A.GND
A/D 12	5	6	A.GND
A/D 13	7	8	A.GND
A/D 14	9	10	A.GND
A/D 15	11	12	A.GND
D/A 1	13	14	A.GND
D/A 2	15	16	A.GND
V.REF 1	17	18	A.GND
V.REF 2	19	20	A.GND

Connector 3 (D/O)

D/O 0	1	2	D/O 1
D/O 2	3	4	D/O 3
D/O 4	5	6	D/O 5
D/O 6	7	8	D/O 7
D/O 8	9	10	D/O 9
D/O 10	11	12	D/O 11
D/O 12	13	14	D/O 13
D/O 14	15	16	D/O 15
D.GND	17	18	D.GND
+5 V	19	20	+12 V

Connector 4 (D/I)

D/I 0	1	2	D/I 1
D/I 2	3	4	D/I 3
D/I 4	5	6	D/I 5
D/I 6	7	8	D/I 7
D/I 8	9	10	D/I 9
D/I 10	11	12	D/I 11
D/I 12	13	14	D/I 13
D/I 14	15	16	D/I 15
D.GND	17	18	D.GND
+5 V	19	20	+12 V

Connector 5 (Counter)

EX.TRG	1	2	
GATE Y	3	4	
2 MHz	5	6	CTR1 GATE
	7	8	CTR0 CLK
	9	10	CTR0 OUT
	11	12	CTR0 GATE
	13	14	CTR1 OUT
	15	16	
D.GND	17	18	D.GND
+5 V	19	20	

PCL-830 10-channel Timer/Counter Card
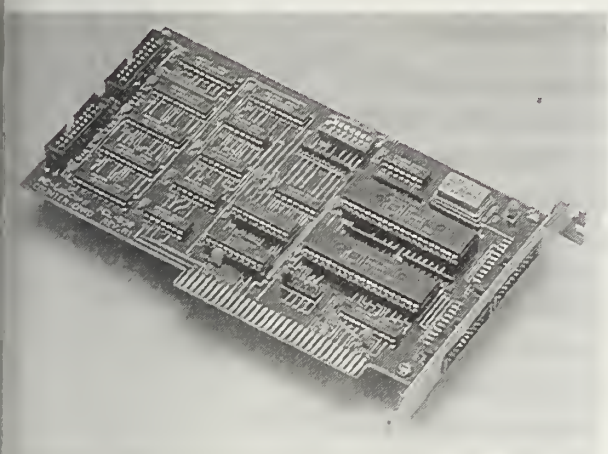
Specifications

Counter

- **Description**: Ten independent 16-bit counters
- **Input level**: TTL-compatible
- **Output level**: TTL-compatible, V_{OL} = 0.4 V max @ 3.2 mA sink; V_{OH} = 2.4 V min @ 0.2 mA source
- **Input frequency**: 6.8 MHz max
- **Input pulse width**: >70 ns
- **Connector**: Two 20-pin flat-cable connectors
- **Time base**: 1.00 MHz
- **Frequency stability**: ±100 PPM

Digital I/O

- **Channels**: 16 TTL-compatible outputs (16 bits)
- **Driving capacity**: Sink 8.0 mA @ 0.5 V (low), source 0.4 mA @ 2.4 V (high)

General

- **Dimensions**: 7" x 4.2" (179 mm x 107 mm)
- **Power consumption**: +5 V @ 600 mA typical

Pin Assignment

CN 1

	1	2	
chip #1 oscillator out	1	2	counter #1 gate
counter #2 gate	3	4	counter #3 gate
Counter #4 gate	5	6	counter #5 gate
counter #1 input	7	8	counter #2 input
counter #3 input	9	10	counter #4 input
counter #5 input	11	12	counter #1 output
counter #2 output	13	14	counter #3 output
counter #4 output	15	16	counter #5 output
ground	17	18	interrupt enable
+5 V power	19	20	interrupt input

CN 2

	1	2	
chip #2 oscillator out	1	2	counter #6 gate
counter #7 gate	3	4	counter #8 gate
counter #9 gate	5	6	counter #10 gate
counter #6 input	7	8	counter #7 input
counter #8 input	9	10	counter #9 input
counter #10 input	11	12	counter #6 output
counter #7 output	13	14	counter #8 output
counter #9 output	15	16	counter #10 output
ground	17	18	not used
+5 V power	19	20	not used

Introduction

PCL-830 is a general purpose counter/timer and digital card for PC/AT compatible computers. It provides ten bit up/down counter channels and frequency dividers for n-board 4 MHz crystal time base. It also includes 16 tal outputs and 16 digital inputs. Two AMD 9513 chips vide a variety of powerful counter/timer function modes to ch your industrial and laboratory applications.

Applications

- vent counting
- dustrial automation (flowmeter and wattmeter monitoring)
- rogrammable frequency synthesis
- equency counter
- ulse-width and period measurement
- me-delay generation
- equency-shift keying
- V conversion and pulse accumulation

Features

- eriodic interrupt generation
- independent 16-bit up/down counters
- inary or BCD counting
- rogrammable frequency output
- omplex duty-cycle output
- wo alarm comparators (on counters #1/#2 and #6/#7)
- ingle-shot or continuous output
- rogrammable count/gate source selection
- ogrammable input and output polarity
- rogrammable gate functions
- 6-bit TTL input and 16-bit TTL output ports
- electable interrupt input channel
- p to 6.8 MHz input frequency
- me-of-day option

Ordering Information

☐ **PCL-830**: 10-channel Counter/Timer Card, user's manual and utility diskette

PCL-838 Stepping Motor Control Card

Specifications

Pulse-train generator
- **Independent channels**: 3
- **Steps per command**: 1 to 65,535
- **Speed range**: From 1 to 7000 pps (pulses per second)
- **Operating modes**: Either two-pulse (CW, CCW) mode or pulse-direction mode, selectable by DIP switch
- **Signals**: Opto-coupled with open collector
- **Pull-up voltage**: +5 V, +12 V, or external
- **Pull-up resistor**: 4.7 KΩ
- **Driving capacity**: 30 mA @ 0.5 V

Digital I/O
- **Input**: 24 channels, TTL compatible
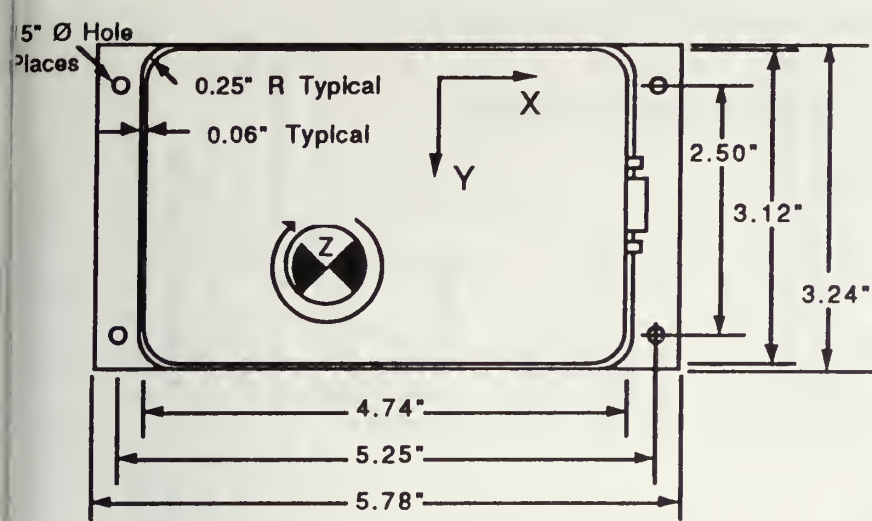- **Output**: 24 channels, TTL compatible

General
- **Dimensions**: 13.3" (L) x 3.8" (W) (340 mm x 98 mm)
- **Power consumption**:
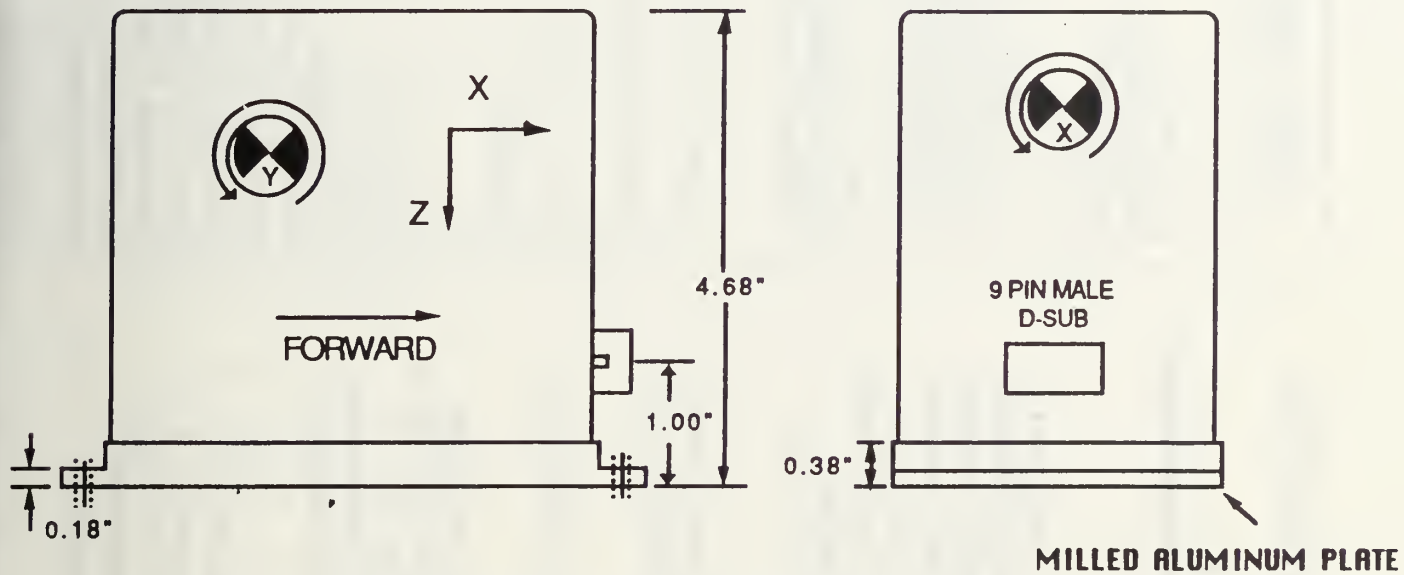 5 V @ 1.2 A typical, 12 V external load only

Introduction

The PCL-838 Stepping Motor Control Card turns your PC into a multi-axis motion-control station. The intelligent PCL-838 fetches operation data from its dual-port RAM to generate pulses for each channel, giving higher performance.

You can install more than one card in your PC, each card controlling up to three motors at the same time. The included DOS device driver provides powerful commands that support you to easily incorporate motion control in your application software.

Applications

- Precise X-Y-Z position control
- Precise rotation control
- Robotics and assembly equipment
- Other stepping-motor applications

Features

- Independent and simultaneous control of up to three motors
- Device driver with a language-independent high-level command interpreter
- Programmable step rate from 1 to 7000 pps (pulses per second)
- Programmable initial speed, final speed and time duration with calculated linear acceleration and deceleration
- Supports one clock (pulse and direction) and two clock (CW pulse and CCW pulse) control modes

Pin assignment

Left	Pin	Pin	Right
COMMON (CH1)	1		
		20	OIR/CW (CH1)
PULSE/CCW (CH1)	2	21	EXT.VCC (CH1)
COMMON (CH2)	3	22	OIR/CW (CH2)
PULSE/CCW (CH2)	4	23	EXT.VCC (CH2)
COMMON (CH3)	5	24	DIR/CW (CH3)
PULSE/CCW (CH3)	6	25	EXT.VCC (CH3)
E.STOP (CH1)	7	26	GND
E.STOP (CH2)	8	27	+12 V
E.STOP (CH3)	9	28	+5 V
GNO	10	29	GNO
GND	11	30	D/O 0
O/I 0	12	31	D/O 1
D/I 1	13	32	D/O 2
D/I 2	14	33	O/O 3
O/I 3	15	34	D/O 4
D/I 4	16	35	D/O 5
D/I 5	17	36	D/O 6
O/I 6	18	37	D/O 7
D/I 7	19		

COMMON: Isolated common point
EXT.VCC: External power source
PULSE/CCW: Stepping pulses or CCW pulses
DIR/CW: Direction signal or CW pulses
E.STOP: Emergency stop
GND: Ground of the PC
D/I 0 to D/I 7: Digital inputs
D/O 0 to D/O 7: Digital outputs

INERTIAL MEASUREMENT UNIT

5" Ø Hole Places

0.25" R Typical

0.06" Typical

X

Y

Z

2.50"

3.12"

3.24"

4.74"

5.25"

5.78"

X

Z

Y

FORWARD

4.68"

1.00"

0.18"

9 PIN MALE D-SUB

X

0.38"

MILLED ALUMINUM PLATE

INERTIAL MEASUREMENT UNIT
9 PIN MALE CONNECTOR

1	POWER GND
2	+28 VDC IN
3	SIGNAL GND
4	
5	RX
6	
7	
8	
9	TX**

**THE USER RECEIVES ON THIS LINE.

WATSON INDUSTRIES, INC.
3041 Melby Road Eau Claire, WI 54703
Phone (715)839-0628 • FAX (715)839-8248

7/09/93

INERTIAL MEASUREMENT UNIT (IMU-600AD)

Watson Industries has provided a variety of sensor packages to customers world wide. We have listened to the many demands for a complete dynamic sensor package which is highly reliable and easy to use. The IMU-600AD combines multi-layer surface mount technology with proven sensor expertise to deliver a complete sensor package - Tri-Axial rate and acceleration, bank and elevation, and magnetic heading - all in one compact package. All data is sampled 50 times a second with 16 bit accuracy.

RATE GYROS - The rate gyros are solid-state, vibrating element angular rate gyros. They combine excellent performance with small size and low power.

ACCELEROMETERS - The accelerometers are instrumentation grade. Alternate ranges are available.

BANK AND ELEVATION - These sensors are liquid capacitive vials which provide excellent repeatibility, resolution, as well as a small size. The bank and elevation sensors provide the reference to calibrate the bias settings for the accelerometers as well as information for Euler coordination transformation on the tri-axial magnetometer data.

MAGNETIC HEADING SENSOR - A tri-axial magnetometer, combined with the bank and elevation sensors, provide accurate magnetic heading data. The data from the roll and pitch sensors provides information for an euler coordinate transformation of magnetic data. This results in accurate heading information without the use of gimbals.

IMU-600AD SPECIFICATIONS

RATE RANGE	±100°/sec
RATE ACCURACY	1% F.S.
RATE RESOLUTION	< 0.05°/SEC
RATE BIAS	±5% F.S.
ACCELERATION RANGE	±3 g's
ACCELERATION ACCURACY	< 0.5% F.S.
ACCELERATION BIAS	< 0.5% F.S.
ACCELERATION RESOLUTION	BETTER THAN 5 mG's
BANK AND ELEV. ACCURACY	0.2° to 30°
SENSOR ALIGNMENT	<0.25° - ALL SENSORS
FREQUENCY RESPONSE	DC-20 Hz (Accel. and Rate)
	DC-0.5 Hz (Roll, Pitch, Heading)
HEADING ACCURACY	±3° to 15° Tilt
POWER REQUIREMENT	+28VDC
	< 250mA
OUTPUT FORMAT	DIGITAL, RS232
(OPTIONAL)	ANALOG
SIZE	SEE DRAWING
WEIGHT	< 2 LBS
SHOCK	50 g's
TEMPERATURE	-20°C to +60°C

SENSOR ALIGNMENT - All sensors are guaranteed not to be misaligned with its proper axis by no more than 0.3°. This feature is critical for a tri-axial set of sensors. The resolution of a sensor can be overwhelmed by corruption of data due to sensor mis-alignment.

CONVENIENCE - You install only one sensor package.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
   Cameron Station
   Alexandria, VA   22304-6145

2. Dudley Knox Library   2
   Code 52
   Naval Postgraduate School
   Monterey, CA   93943-5101

3. Chairman, Code EC   1
   Electrical and Computer Engineering Department
   Naval Postgraduate School
   Monterey, CA   93943-5121

4. Chairman, Code CS   1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA   93943-5118

5. Dr. Michael K. Shields, Code EC/MS   2
   Electrical and Computer Engineering Department
   Naval Postgraduate School
   Monterey, CA   93943-5121

6. Dr. Se-Hung Kwak   2
   Loral, Inc.
   Advanced Distributed Simulation
   50 Moulton Street
   Cambridge, MA   02138

7. Commandant (G-EAE)   1
   U.S. Coast Guard Headquarters
   ATTN: LCDR Pat Moran
   2100 2nd Street S.W.
   Washington, DC   20593

8. Commanding Officer   2
   U.S. Coast Guard Research and Development Center
   ATTN: ENS Carlton Williams
   Avery Point
   Groton, CT   06340-6096

9. LT Peter M. Hoffman   1
   63 Lexington Drive
   Croton-on-Hudson, NY   10520